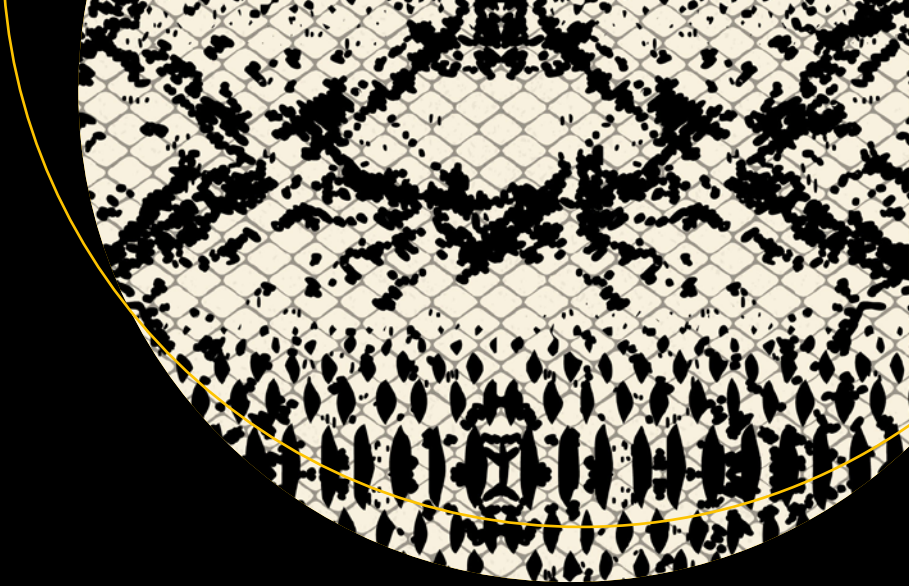




Beginning Anomaly Detection Using Python-Based Deep Learning

With Keras and PyTorch

Sridhar Alla
Suman Kalyan Adari

Apress®

Beginning Anomaly Detection Using Python-Based Deep Learning

With Keras and PyTorch

**Sridhar Alla
Suman Kalyan Adari**

Apress®

Beginning Anomaly Detection Using Python-Based Deep Learning: With Keras and PyTorch

Sridhar Alla
New Jersey, NJ, USA

Suman Kalyan Adari
Tampa, FL, USA

ISBN-13 (pbk): 978-1-4842-5176-8
<https://doi.org/10.1007/978-1-4842-5177-5>

ISBN-13 (electronic): 978-1-4842-5177-5

Copyright © 2019 by Sridhar Alla, Suman Kalyan Adari

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: Laura Berendson
Coordinating Editor: Aditee Mirashi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-5176-8. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

- About the Authors..... ix
- About the Technical Reviewers xi
- Acknowledgments xiii
- Introduction xv
- Chapter 1: What Is Anomaly Detection? 1
 - What Is an Anomaly?..... 1
 - Anomalous Swans 1
 - Anomalies as Data Points 5
 - Anomalies in a Time Series 9
 - Taxi Cabs 11
 - Categories of Anomalies 15
 - Data Point-Based Anomalies 16
 - Context-Based Anomalies..... 16
 - Pattern-Based Anomalies 17
 - Anomaly Detection 17
 - Outlier Detection..... 18
 - Noise Removal..... 18
 - Novelty Detection 18
 - The Three Styles of Anomaly Detection 19
 - Where Is Anomaly Detection Used? 20
 - Data Breaches 20
 - Identity Theft..... 21
 - Manufacturing 21

TABLE OF CONTENTS

Networking	22
Medicine	22
Video Surveillance	23
Summary.....	23
Chapter 2: Traditional Methods of Anomaly Detection	25
Data Science Review	25
Isolation Forest	34
Mutant Fish.....	34
Anomaly Detection with Isolation Forest.....	36
One-Class Support Vector Machine.....	51
Anomaly Detection with OC-SVM	63
Summary.....	71
Chapter 3: Introduction to Deep Learning.....	73
What Is Deep Learning?	73
Artificial Neural Networks	74
Intro to Keras: A Simple Classifier Model	84
Intro to PyTorch: A Simple Classifier Model.....	111
Summary.....	122
Chapter 4: Autoencoders	123
What Are Autoencoders?	123
Simple Autoencoders	125
Sparse Autoencoders	140
Deep Autoencoders	142
Convolutional Autoencoders.....	144
Denoising Autoencoders	153
Variational Autoencoders	163
Summary.....	178

Chapter 5: Boltzmann Machines.....	179
What Is a Boltzmann Machine?.....	179
Restricted Boltzmann Machine (RBM)	181
Anomaly Detection with the RBM - Credit Card Data Set.....	187
Anomaly Detection with the RBM - KDDCUP Data Set.....	197
Summary.....	212
Chapter 6: Long Short-Term Memory Models.....	213
Sequences and Time Series Analysis.....	213
What Is a RNN?	216
What Is an LSTM?	218
LSTM for Anomaly Detection.....	223
Examples of Time Series.....	243
art_daily_no_noise.....	243
art_daily_nojump	244
art_daily_jumpsdown.....	246
art_daily_perfect_square_wave	248
art_load_balancer_spikes.....	250
ambient_temperature_system_failure.....	251
ec2_cpu_utilization	253
rds_cpu_utilization.....	254
Summary.....	256
Chapter 7: Temporal Convolutional Networks	257
What Is a Temporal Convolutional Network?.....	257
Dilated Temporal Convolutional Network	262
Anomaly Detection with the Dilated TCN	267
Encoder-Decoder Temporal Convolutional Network.....	283
Anomaly Detection with the ED-TCN	286
Summary.....	295

Chapter 8: Practical Use Cases of Anomaly Detection..... 297

 Anomaly Detection 298

 Real-World Use Cases of Anomaly Detection..... 299

 Telecom 300

 Banking 302

 Environmental 303

 Healthcare 304

 Transportation 306

 Social Media 307

 Finance and Insurance 308

 Cybersecurity..... 309

 Video Surveillance 312

 Manufacturing 313

 Smart Home..... 315

 Retail 315

 Implementation of Deep Learning-Based Anomaly Detection 316

 Summary..... 317

Appendix A: Intro to Keras..... 319

 What Is Keras? 319

 Using Keras 320

 Model Creation 321

 Model Compilation and Training 322

 Model Evaluation and Prediction 326

 Layers..... 328

 Loss Functions..... 340

 Metrics 343

 Optimizers 345

 Activations 348

 Callbacks 351

 Back End (TensorFlow Operations)..... 358

 Summary..... 360

Appendix B: Intro to PyTorch	361
What Is PyTorch?.....	361
Using PyTorch	362
Sequential vs. ModuleList	376
Layers.....	377
Loss Functions.....	387
Optimizers	390
Temporal Convolutional Network in PyTorch.....	392
Dilated Temporal Convolutional Network.....	393
Summary.....	408
Index.....	409

About the Authors



Sridhar Alla is the co-founder and CTO of Bluewhale, which helps organizations big and small in building AI-driven big data solutions and analytics. He is a published author of books and an avid presenter at numerous Strata, Hadoop World, Spark Summit, and other conferences. He also has several patents filed with the US PTO on large-scale computing and distributed systems. He has extensive hands-on experience in several technologies including Spark, Flink, Hadoop, AWS, Azure, Tensorflow, Cassandra,

and others. He spoke on anomaly detection using deep learning at Strata SFO in March 2019 and will also present at Strata London in October 2019.

Sridhar was born in Hyderabad, India and now lives in New Jersey with his wife, Rosie, and daughter, Evelyn. When he is not busy writing code he loves to spend time with his family; he also loves training, coaching, and organizing meetups.

He can be reached via email at sid@bluewhale.one or via LinkedIn at www.linkedin.com/in/sridhar-a-1619b42/. Please visit www.bluewhale.one for more details on how he could help your organization.



Suman Kalyan Adari is an undergraduate student pursuing a B.S. in Computer Science at the University of Florida. He has been conducting deep learning research in the field of cybersecurity since his freshman year, and has presented at the IEEE Dependable Systems and Networks workshop on Dependable and Secure Machine Learning held in Portland, Oregon, USA in June 2019.

He is quite passionate about deep learning, and specializes in its practical uses in various fields such as video processing, image recognition, anomaly detection, targeted adversarial attacks, and more.

He can be contacted via email at sumank.adari@yahoo.com or at sadari@ufl.edu. He also has a LinkedIn account at www.linkedin.com/in/suman-kalyan-adari/.

About the Technical Reviewers



Jojo Moolayil is an artificial intelligence professional and published author of three books on machine learning, deep learning, and IoT. He is currently working with Amazon Web Services as a Research Scientist – A.I. in their Vancouver, BC office.

He was born and raised in Pune, India and graduated from the University of Pune with a major in Information Technology Engineering. His passion for problem solving and data-driven decision making led him to start a career with Mu Sigma Inc., the world's largest pure-play analytics provider, where he was responsible for developing machine learning and decision science solutions for large complex problems for healthcare and telecom giants. He later worked with Flutura (an IoT Analytics startup) and General Electric with a focus on industrial A.I in Bangalore, India.

In his current role with AWS, he works on researching and developing large scale A.I. solutions for combating fraud and enriching the customer's payment experience in the cloud. He is also actively involved as a tech reviewer and AI consultant with leading publishers and has reviewed over a dozen books on machine learning, deep learning, and business analytics.

You can reach Jojo at

- www.jojomoolayil.com/
- www.linkedin.com/in/jojo62000
- <https://twitter.com/jojo62000>

ABOUT THE TECHNICAL REVIEWERS



Satyajit Pattnaik is a Senior Data Scientist with around eight years of expertise in the field. He has a passion for turning data into actionable insights and meaningful stories. Right from the data extraction until the final data product or actionable insights, he enjoys the journey with the data.

He is a dedicated and determined person who can adapt to any environment, which is quite evident from the cross-domain projects involving different types of data, platforms, and techniques he has worked on. Apart from the skills related to data capture, analysis, and presentation, he possesses good problem solving skills. Being from the computer science field is really an add-on to do things quickly and in a reusable manner. Along with machine learning, he believes in quick learning as and when needed.

Acknowledgments

Sridhar Alla

I would like to thank my wonderful, loving wife, Rosie Sarkaria, and my beautiful, loving daughter, Evelyn, for all their love and patience during the many months I spent writing this book. I would also like to thank my parents, Ravi and Lakshmi Alla, for their blessings and all the support and encouragement they continue to bestow upon me.

Suman Kalyan Adari

I would like to thank my parents, Krishna and Jyothi, and my loving dog, Pinky, for supporting me throughout the entire process of writing my first book. I would especially like to thank my sister, Niha, for helping me with graph creation, proof-reading, editing, and testing the code samples.

Introduction

Congratulations on your decision to explore deep learning and the exciting world of anomaly detection using deep learning.

Anomaly detection is finding patterns that do not adhere to what is considered as normal or expected behavior. Businesses could lose millions of dollars due to abnormal events. Consumers could also lose millions of dollars. In fact, there are many situations every day where people's lives are at risk and where their property is at risk. If your bank account gets cleaned out, that is a problem. If your water line breaks, flooding your basement, that's a problem. If all flights get delayed in the airport, causing long delays, that's a problem. You might have been misdiagnosed or not diagnosed at all with a health issue, which is a very big problem directly impacting your well-being.

In this book, you will learn how anomaly detection can be used to solve business problems. You will explore how anomaly detection techniques can be used to address practical use cases and address real-life problems in the business landscape. Every business and use case is different, so while we cannot copy-paste code and build a successful model to detect anomalies in any dataset, this book will cover many use cases with hands-on coding exercises to give an idea of the possibilities and concepts behind the thought process.

We choose Python because it is truly the best language for data science with a plethora of packages and integrations with scikit-learn, deep learning libraries, etc.

We will start by introducing anomaly detection and then we will look at legacy methods of detecting anomalies used for decades. Then we will look at deep learning to get a taste of it.

Then we will explore autoencoders and variational autoencoders, which are paving the way for the next generation of generative models.

We will explore RBM (Boltzmann machines) as way to detect anomalies. Then we'll look at LSTMs (long short-term memory) models to see how temporal data can be processed.

We will cover TCN (Temporal Convolutional Networks), which are the best in class for temporal data anomaly detection. Finally, we will look at several examples of anomaly detection in various business use cases.

INTRODUCTION

In addition, we will also cover Keras and PyTorch, the two most popular deep learning frameworks in detail in the Appendix chapters.

You will combine all this extensive knowledge with hands-on coding using Jupyter notebook-based exercises to experience the knowledge first hand and see where you can use these algorithms and frameworks.

Best of luck and welcome to the world of deep learning!

CHAPTER 1

What Is Anomaly Detection?

In this chapter, you will learn about anomalies in general, the categories of anomalies, and anomaly detection. You will also learn why anomaly detection is important and how anomalies can be detected and the use case for such a mechanism.

In a nutshell, the following topics will be covered throughout this chapter:

- What is an anomaly?
- Categories of different anomalies
- What is anomaly detection?
- Where is anomaly detection used?

What Is an Anomaly?

Before you get started with learning about anomaly detection, you must first understand exactly what you are targeting. Generally, an **anomaly** is an outcome or value that deviates from what is expected, but the exact criteria for what determines an anomaly can vary from situation to situation.

Anomalous Swans

To get a better understanding of what an anomaly is, let's take a look at some swans sitting by a lake (Figure 1-1).

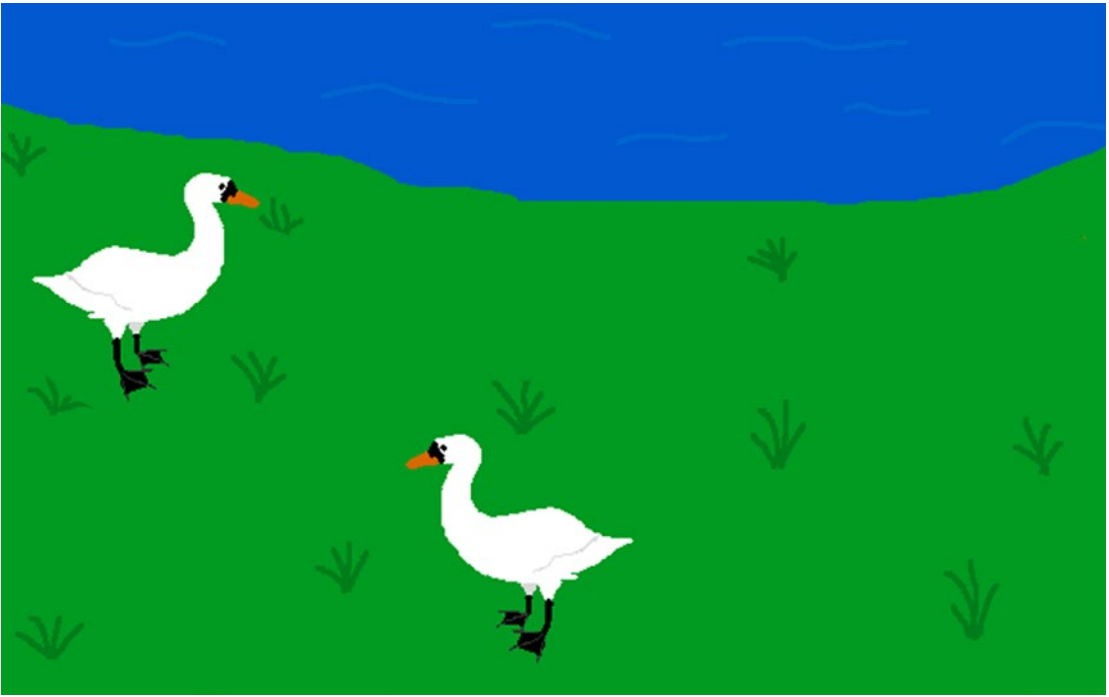


Figure 1-1. *A couple of swans by a lake*

Say you want to observe these swans and make assumptions about the color of the swans. Your goal is to determine the normal color of swans and to see if there are any swans that are of a different color than this (Figure 1-2).

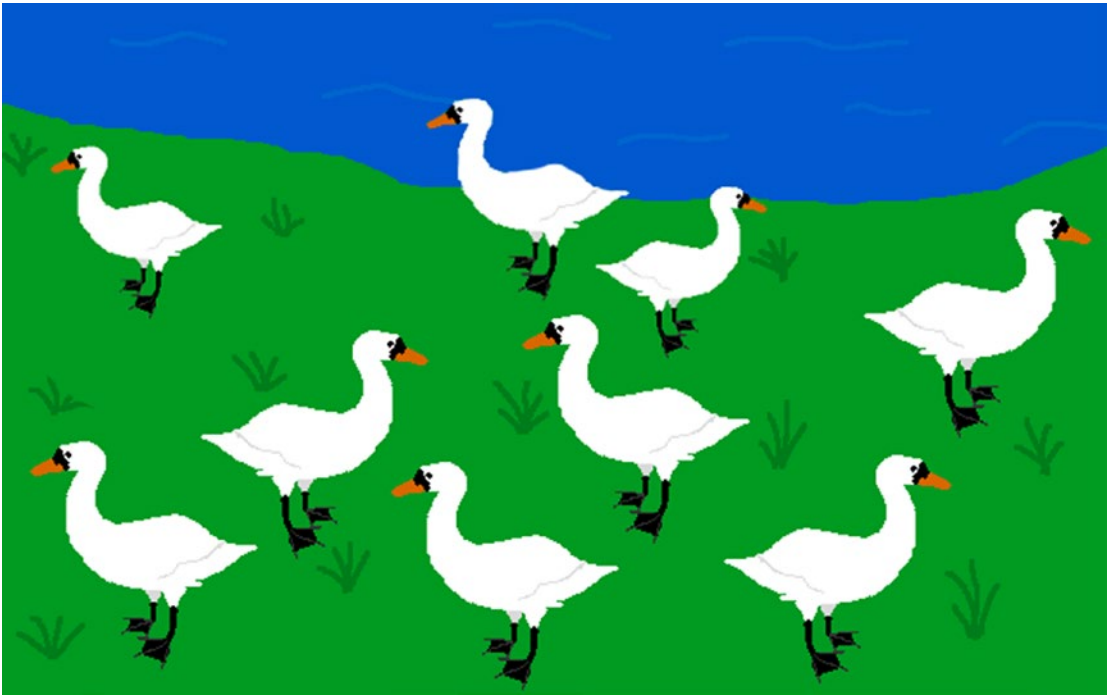


Figure 1-2. *More swans show up, and they're all white swans*

More swans show up, and given that you haven't seen any swans that aren't white, it seems reasonable to assume that all swans at this lake are white. Let's just keep observing these swans, shall we?

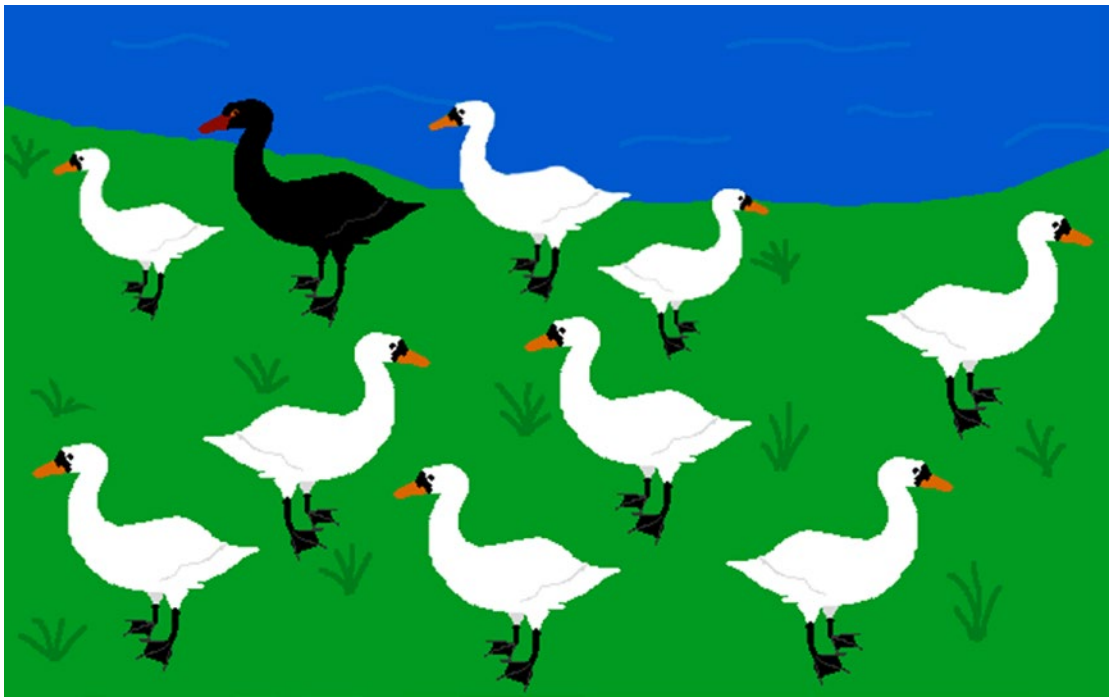


Figure 1-3. *A black swan appears*

What's this? Now you see a black swan show up (Figure 1-3), but how can this be? Considering all of your previous observations, you've seen enough of the swans to assume that the next swan would also be white. However, the black swan you see defies that entirely, making it an anomaly. It's not really an outlier where you could have a really big white swan or really small white swan, but it's a swan that's entirely a different color, making it the anomaly. In this scenario, the overwhelming majority of swans are white, making the black swan extremely rare.

In other words, given a swan by the lake, the probability of it being black is very small. You can explain your reasoning for labeling the black swan as an anomaly with one of two approaches, though you aren't just limited to these two approaches.

First, given that a vast majority of swans observed at this particular lake are white, you can assume that, through a process similar to inductive reasoning, the normal color for a swan here is white. Naturally, you would label the black swan as an anomaly purely based on your prior assumption that all swans are white, considering that you've only seen white swans thus far.

Another way to look at why the black swan is an anomaly is through probability. Assuming that there is a total of 1000 swans at this giant lake with only two black swans,

the probability of a swan being black is $2/1000$, or 0.002 . Depending on the probability threshold, meaning the lowest probability for an outcome or event that will be accepted as normal, the black swan could be labeled as anomalous or normal. In your case, you will consider it an anomaly because of its extreme rarity at this lake.

Anomalies as Data Points

Let's extend this same concept to a real-world application. In the following example, you will take a look at a factory that produces screws and attempt to determine what an anomaly could be in this context. The factory produces massive batches of screws all at once, and samples from each batch are tested to ensure that a certain level of quality is maintained. For each sample, assume that the density and tensile strength (how resistant the screw is to breaking under stress) is measured.

Figure 1-4 is an example graph of various sample batches with the dotted lines representing the range of densities and tensile strengths allowed.

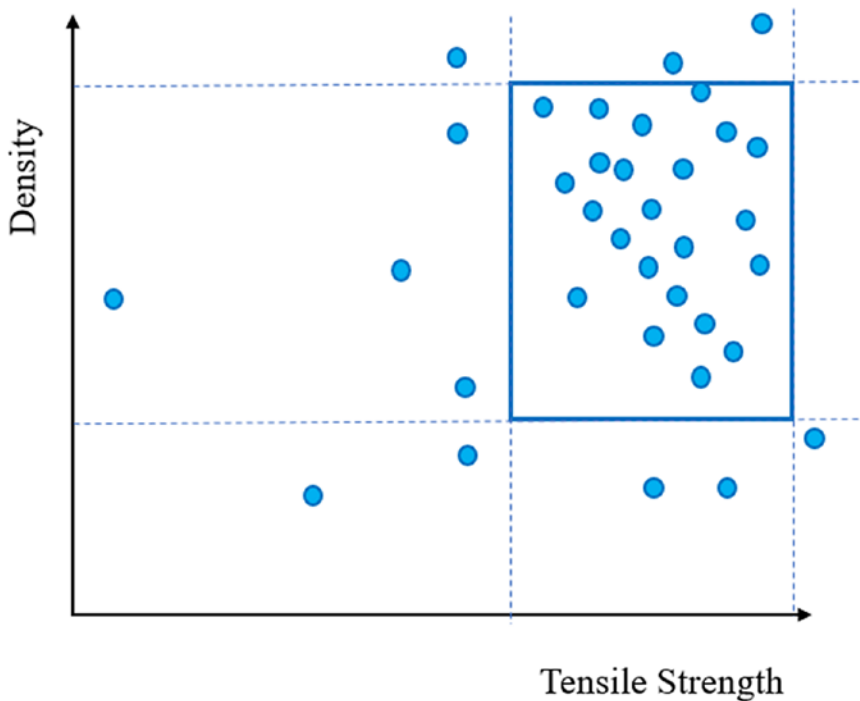


Figure 1-4. *Density and tensile strength in sample batches of screws*

The intersections of the dotted lines create several different regions containing data points. Of interest is the bounding box (solid lines) created from the intersection of both dotted lines since it contains the data points for samples deemed acceptable (Figure 1-5). Any data point outside of that specific box will be considered anomalous.

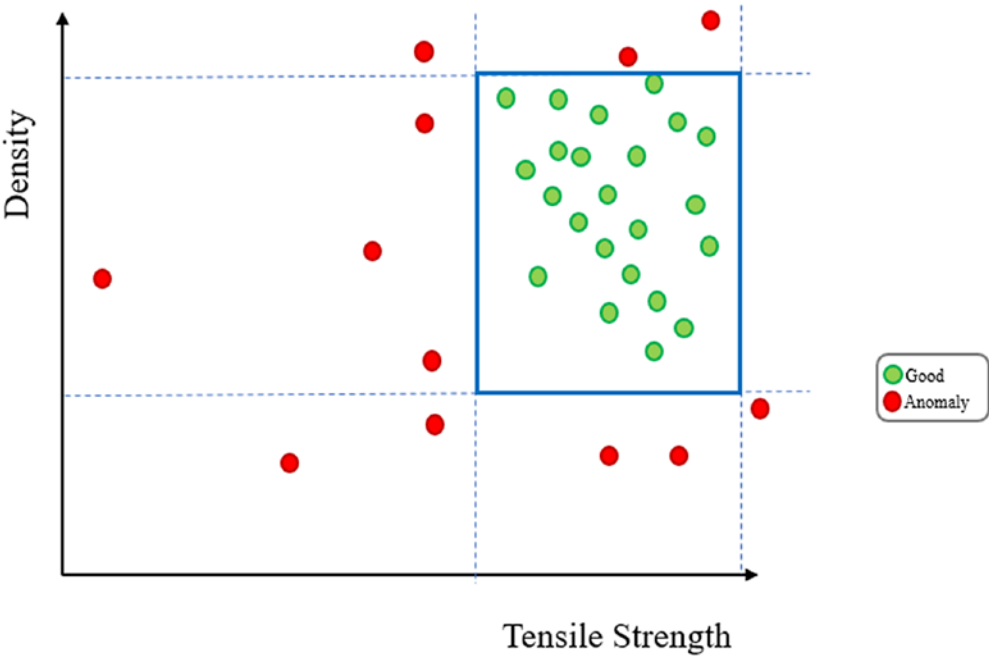


Figure 1-5. Data points are identified as good or anomaly based on their location

Now that you know what points are and aren’t acceptable, let’s pick out a sample from a new batch of screws and check its data to see where it falls on the graph (Figure 1-6).

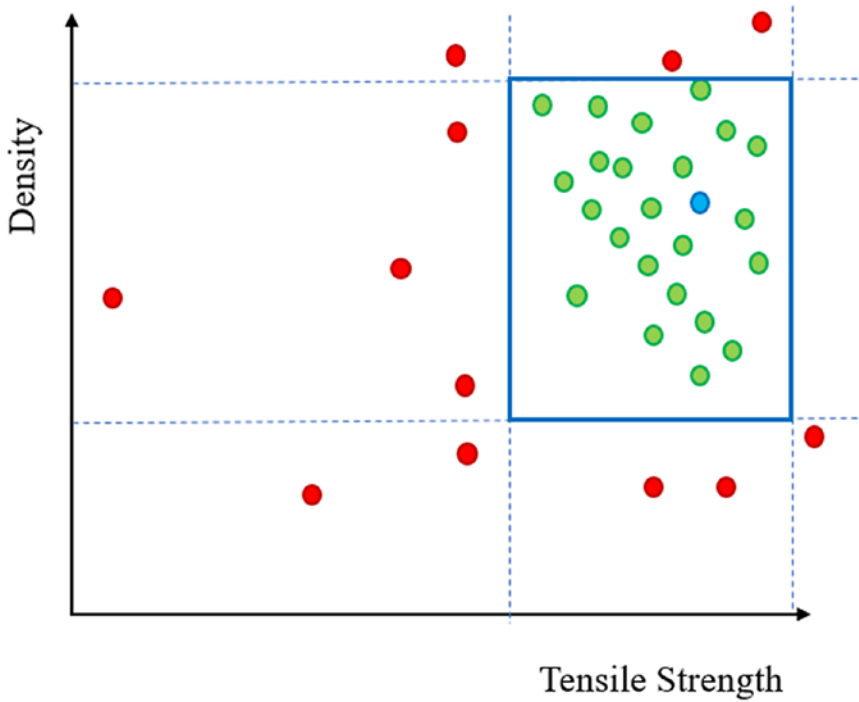


Figure 1-6. A new data point representing the new sample screw is generated, with the data falling within the bounding box

The data for this sample screw falls within the acceptable range. That means that this batch of screws is good to use since its density and tensile strength are appropriate for use by the consumer. Now let's look at a sample from the next batch of screws and check its data (Figure 1-7).

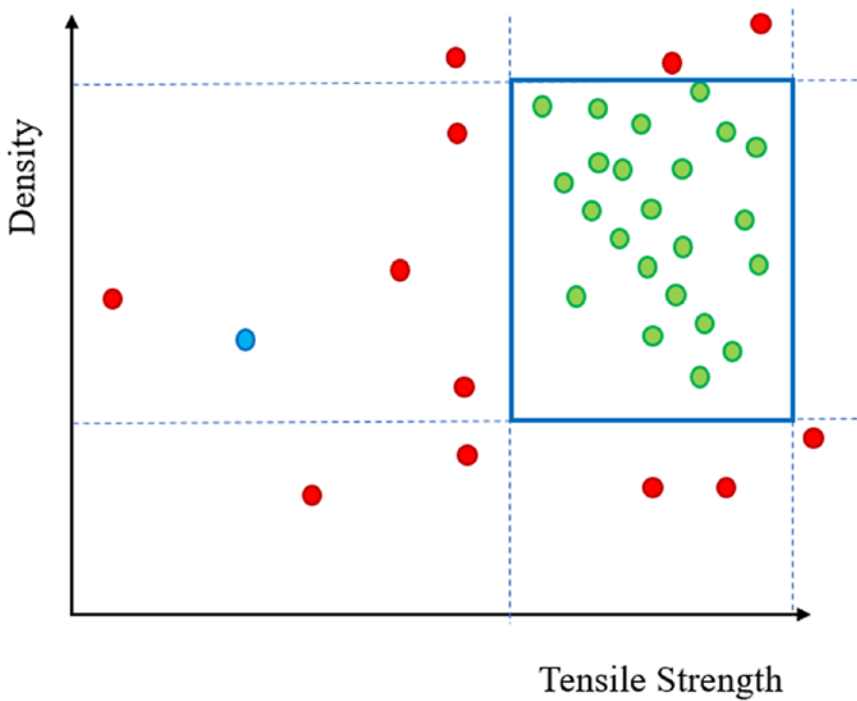


Figure 1-7. A new data point is generated for another sample, but it falls outside the bounding box

The data falls far outside the acceptable range. For its density, the screw has abysmal tensile strength and is unfit for use. Since it has been flagged as an anomaly, the factory can investigate the reasons for why this batch of screws turned out to be brittle. For a factory of considerable size, it is important to hold a high standard of quality as well as maintain a high volume of steady output to keep up with consumer demand. For a monumental task like that, automation to detect any anomalies to avoid sending out faulty screws is essential and has the benefit of being extremely scalable.

So far, you have explored anomalies as data points that are either out of place, in the case of the black swan, or unwanted, in the case of faulty screws. So what happens when you introduce time as a new variable?

Anomalies in a Time Series

With the introduction of time as a variable, you are now dealing with a notion of temporality associated with the data sets. What this means is that certain patterns can emerge based on the time stamp, so you can see monthly occurrences of some phenomenon.

To better understand time-series based anomalies, let's take a random person and look into his/her spending habits over some arbitrary month (Figure 1-8).

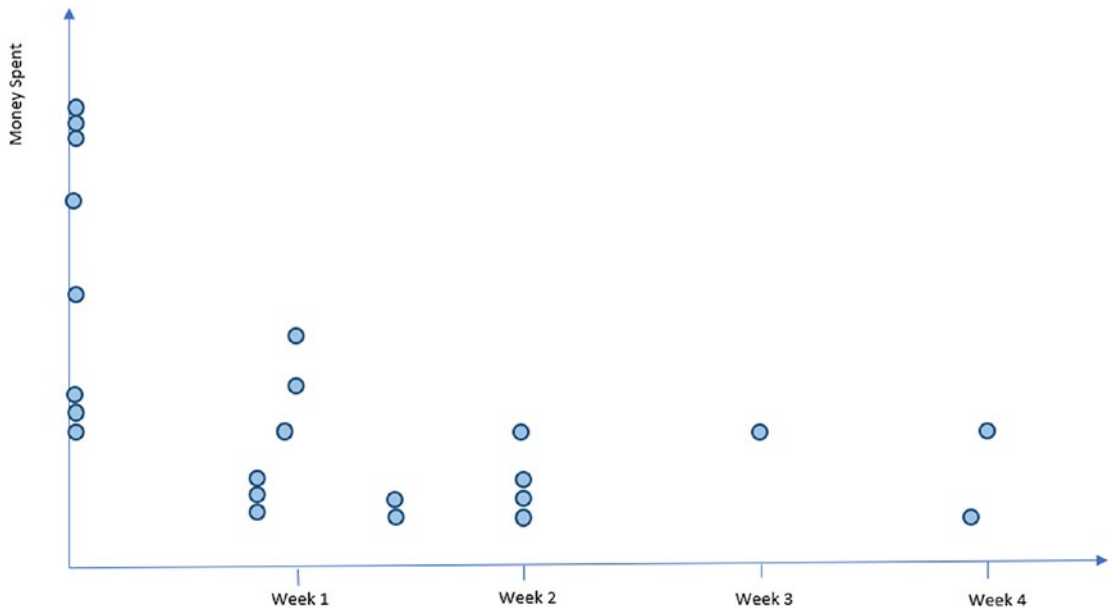


Figure 1-8. *Spending habits of a person over the course of a month*

Assume the initial spike in expenditures at the start of the month is due to the payment of bills like rent and insurance. During the weekdays, our person occasionally eats out, and on the weekends goes shopping for groceries, clothes, or just various items.

These expenditures can vary from month to month from the influence of various holidays. Let's take a look at November, when you can expect a massive spike in purchases on Black Friday (Figure 1-9).

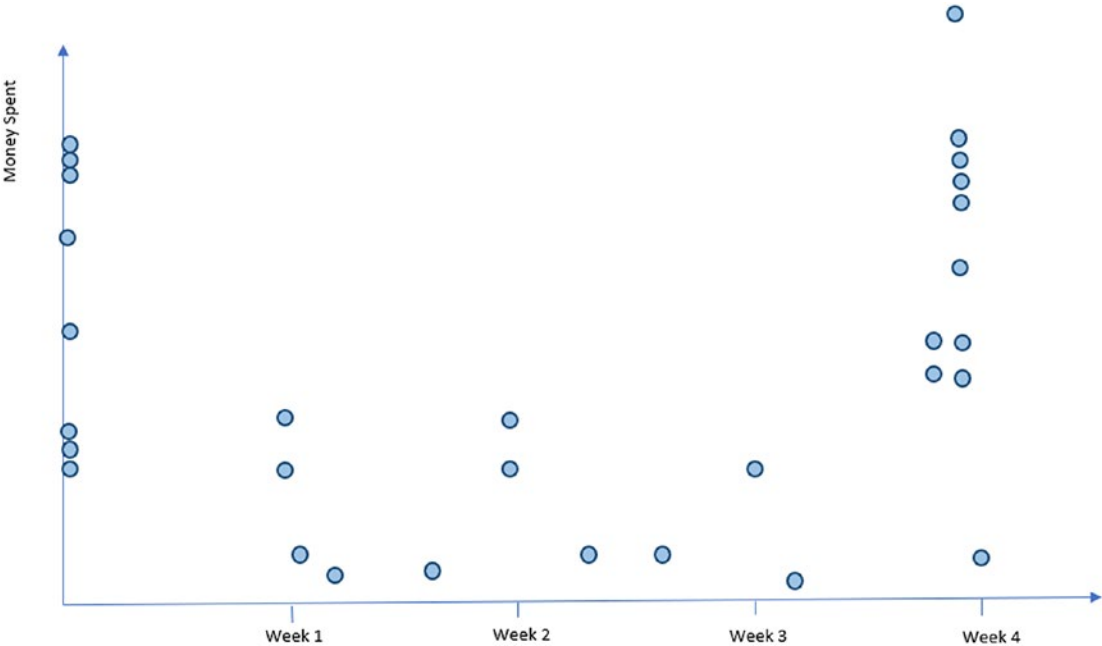


Figure 1-9. *Spending habits for the same person during the month of November*

As expected, there are a lot of purchases made on Black Friday, some of them quite expensive. However, this spike is expected since it is a common trend for many people. Now assume that unfortunately, your person had his/her credit card information stolen, and the criminals responsible for it have decided to purchase various items of interest to them. Using the same month as in the first example (Figure 1-8), Figure 1-10 is a possible graph showcasing what could happen.

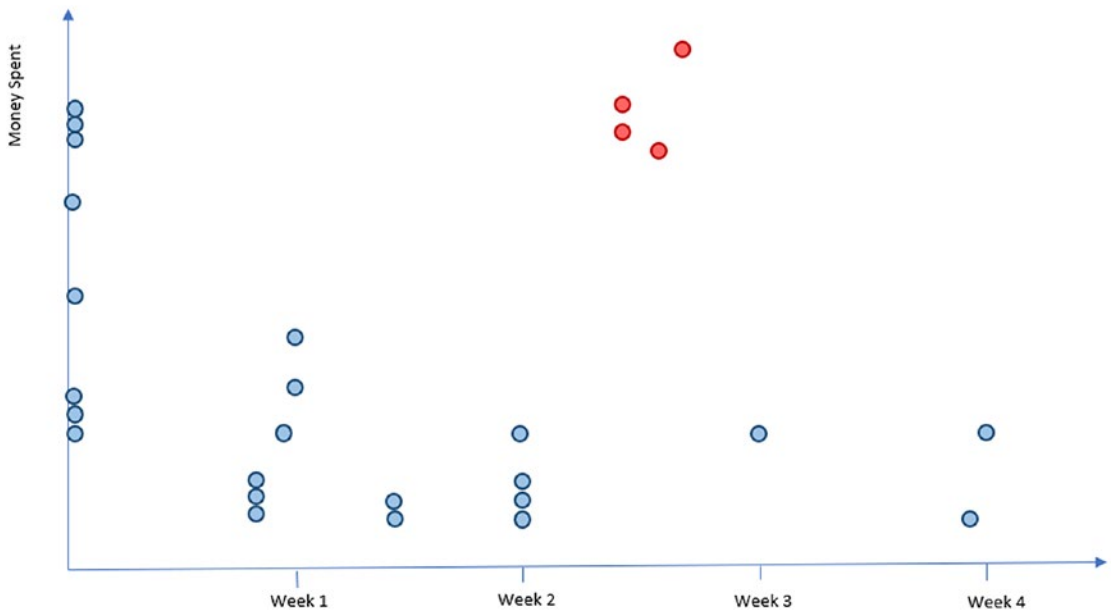


Figure 1-10. Graph of purchases for the person during the same month as in Figure 1-8

Because of the record of purchases for the user from a previous year, the sudden influx in purchases would be flagged as anomalies given the context. Such a cluster of purchases might be normal for Black Friday or before Christmas, but in any other month without a major holiday it might look out of place. In this case, your person might be contacted by the corresponding officials to confirm if they made the purchase or not.

Some companies might even flag purchases that follow normal societal trends. What if that TV wasn't really bought by your person on Black Friday? In that case, company software can ask the client directly through a phone app, for example, whether or not he/she actually bought the item in question, allowing for some additional protection against fraudulent purchases.

Taxi Cabs

Similarly, you can look at the data for taxi cab pickups and drop-offs over time for a random city and see if you can detect any anomalies. On an average day, the total number of pickups can look somewhat like Figure 1-11.

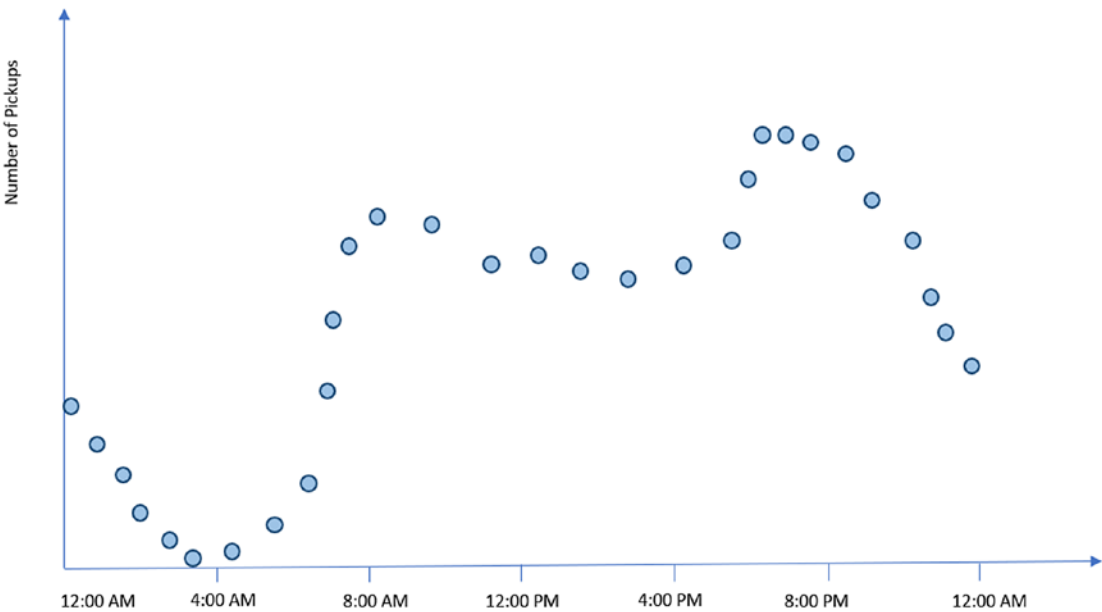


Figure 1-11. *Graph of the number of pickups for a taxi company throughout the day*

From the graph, you see that there’s a bit of post-midnight activity that drops off to near nothing during the late-night hours. However, it picks up suddenly around morning rush hour and remains high until the evening, when it peaks during evening rush hour. This is essentially what an average day looks like.

Let’s expand the scope out a bit more to gain some perspective of passenger traffic throughout the week; see [Figure 1-12](#).

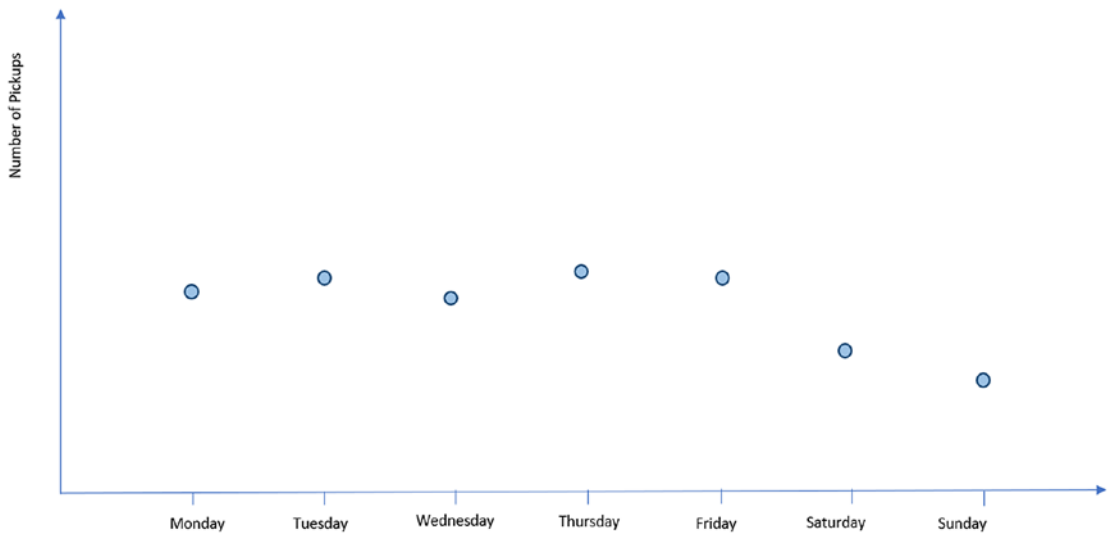


Figure 1-12. *Graph of the number of pickups for a taxi company throughout the week*

As expected, most of the pickups occur during the weekday when commuters must get to and from work. On the weekends, a fair amount of people still go out to get groceries or just go out somewhere for the weekend.

On a small scale like this, causes for anomalies are anything that prevents taxis from operating or incentivizes customers not to use a taxi. For example, say that a terrible thunderstorm hits on Friday. Figure 1-13 shows that graph.

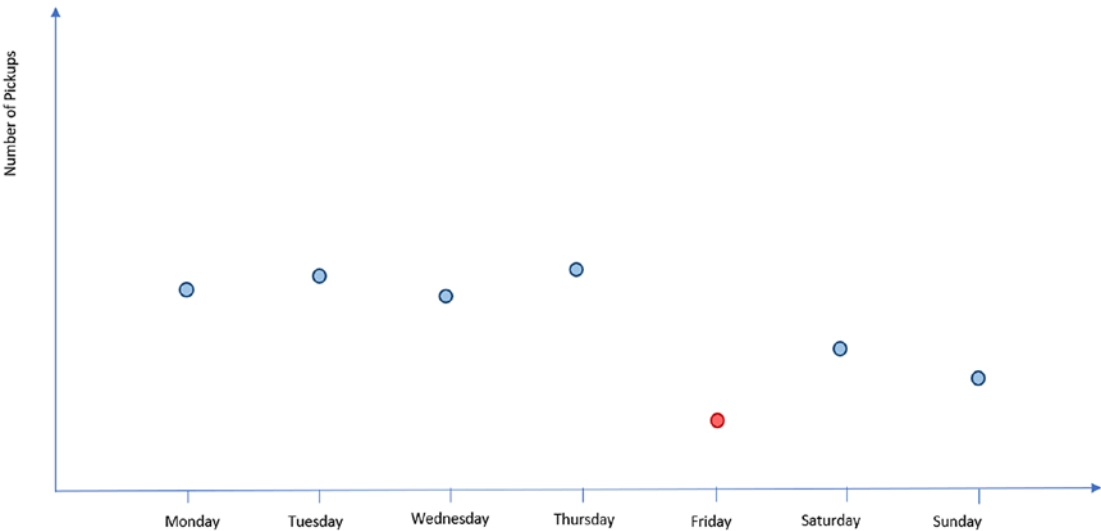


Figure 1-13. Graph of the number of pickups for a taxi company throughout the week, with a heavy thunderstorm on Friday

The presence of the thunderstorm could have influenced some people to stay indoors, resulting in a lower number of pickups than usual for a weekday. However, these sorts of anomalies are usually too small scale and to have any noticeable effect on the overall pattern.

Let’s take a look at the data over the entire year; see Figure 1-14.

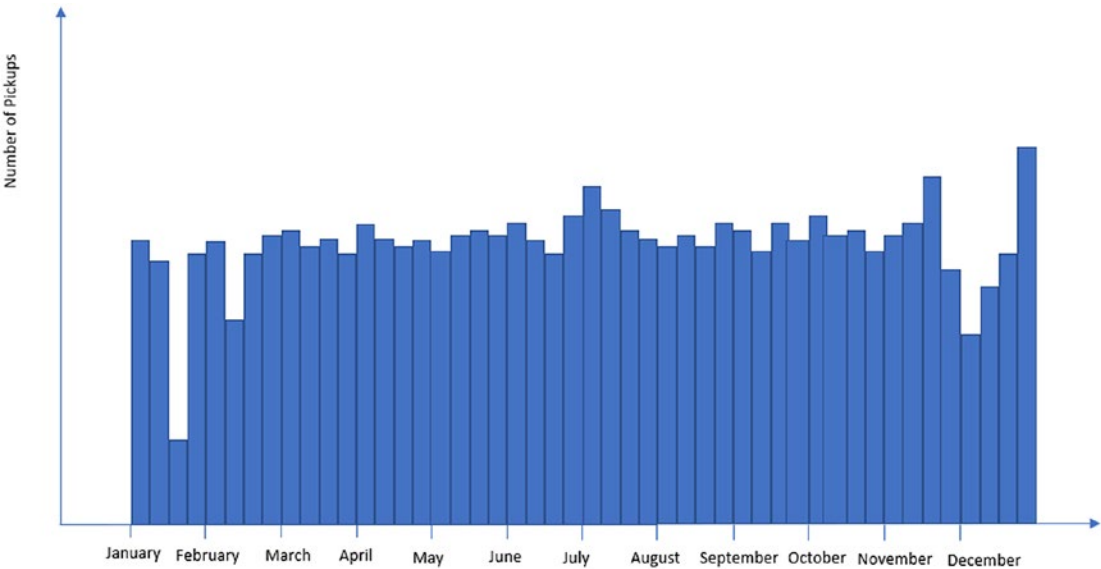


Figure 1-14. Number of pickups for a taxi company throughout the year

The dips occur around the winter months when snowstorms are expected. Sure enough, these are regular patterns that can be observed at similar times every year, so they are not an anomaly. But what happens when a polar vortex descends sometime in April?

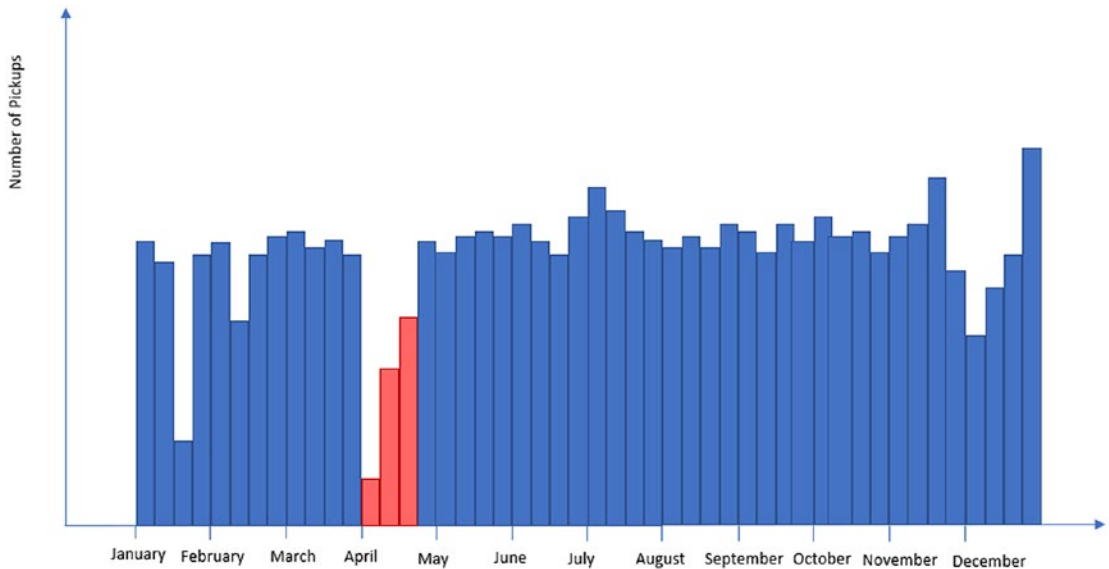


Figure 1-15. Number of pickups for a taxi company throughout the year, with a polar vortex hitting the city in April

As you can see in Figure 1-15, the vortex unleashes several intense blizzards on the imaginary city, severely slowing down all traffic in the first week and burdening the city in the following two weeks. Comparing this graph from the one above, there's a clearly defined anomaly in the graph caused by the polar vortex for the month of April. Since this pattern is extremely rare for the month of April, it would be flagged as an anomaly.

Categories of Anomalies

Now that you have some perspective of what anomalies can be in various situations, you can see that they generally fall into these broad categories:

- Data point-based anomalies
- Context-based anomalies
- Pattern-based anomalies

Data Point-Based Anomalies

Data point-based anomalies can seem comparable to outliers in a set of data points. However, anomalies and outliers are not the same thing. **Outliers** are data points that are expected to be present in the data set and can be caused by unavoidable random errors or from systematic errors relating to how the data was sampled. **Anomalies** are outliers or other values that one doesn't expect to exist. These types of anomalies can be found wherever a data set of values exists.

An example of this is a data set of thyroid diagnostic values, where the majority of the data points are indicative of normal thyroid functionality. In this case, anomalous values represent sick thyroids. While they are not necessarily outliers, they have a low probability of existing when taking into account all the normal data.

You can also detect individual purchases totaling to excessive amounts and label them as anomalies since, by definition, they are not expected to occur or have a very low probability of occurrence. In this case, they are labeled as fraud transactions, and the card holder is contacted to ensure the validity of the purchase.

Basically, you can say this about the difference between anomalies and outliers: you should expect there to be outliers in a set of data, but not anomalies.

Context-Based Anomalies

Context-based anomalies consist of data points that might seem normal at first, but are considered anomalies in their respective contexts. For example, you might expect a sudden surge in purchases near certain holidays, but these purchases could seem out of place in the middle of August. As you saw in the example earlier, the person who made a high volume of purchases towards Black Friday was not flagged because it is typical for people to do so around that time. However, if the purchases were made in a month where it is out of place given previous purchase history, it would be flagged as an anomaly. This might seem similar to the example brought up for data point-based anomalies; the distinction here is that the individual purchase does not have to be expensive. If your person never buys gasoline because he/she owns an electric car, sudden purchases of gasoline would be out of place given the context. Buying gasoline is quite a normal thing to do for everyone, but in this context, it is an anomaly.

Pattern-Based Anomalies

Pattern-based anomalies are patterns and trends that deviate from their historical counterparts. In the taxi cab example, the pickup counts for the month of April were pretty consistent with the rest of the year. However, once the polar vortex hit, the numbers tanked visibly, defining a huge drop in the graph that was labeled as an anomaly.

Similarly, when monitoring network traffic in the workplace, there are expected patterns of network traffic that are formed from constant monitoring of data over several months or even years for some companies. When an employee attempts to download or upload large volumes of data, it will generate a certain pattern in the overall network traffic flow that could be considered anomalous if it deviates from the employee's usual behavior.

If an external hacker decided to DDOS the company's website (**DDOS**, or a distributed denial-of-service attack, is an attempt to overwhelm the server that handles network flow to a certain website in an attempt to bring the entire website down or stop its functionality), every single attempt would register as an unusual spike in network traffic. All of these spikes are clearly deviants from normal traffic and would be considered anomalous.

Anomaly Detection

With a better understanding of the different types of anomalies you can encounter, you can now proceed to start creating models to detect them. Before you do that, there are a couple approaches you can take, although you are not limited to just these methods.

Recall the reasoning for labeling the swan as an anomaly. One of the reasons was that since all the swans you saw thus far were white, the black swan was the anomaly. Another reason was that since the probability of a swan being black was very low, it was an anomaly since you didn't expect that outcome.

The anomaly detection models you will explore in this book will follow these approaches by either training on normal data to classify anomalies, or classifying anomalies by their probabilities if they are below a certain threshold. However, in one of the classes of models that you choose, the anomalies and normal data points will both be labeled as such, so you will basically be told what swans are normal and what swans are anomalies.

Finally, let's explore anomaly detection. **Anomaly detection** is the process in which an advanced algorithm identifies certain data or data patterns to be anomalous. Heavily related to anomaly detection are the tasks of outlier detection, noise removal, and novelty detection. In this book, you will explore all of these options as they are all basically anomaly detection methods.

Outlier Detection

Outlier detection is a technique that aims to detect anomalous outliers within a given data set. As discussed, three methods that can be applied to this situation are to train only on normal data to identify anomalies by a high reconstruction error, to model a probability distribution in which anomalies are labeled based on their association with really low probabilities, or to train a model to recognize anomalies by teaching it what an anomaly looks like and what a normal point looks like.

Regarding the high reconstruction error, think of the model as having trouble labeling an anomaly because it is odd compared to all the normal data points that it has seen. Just like how the black swan is really different based on your initial assumption that all swans are white, the model perceives this anomalous data point as “different” and has a harder time interpreting it.

Noise Removal

In **noise removal**, there is constant background noise in the data set that must be filtered out. Imagine that you are at a party and you are talking to your friend. There is a lot of background noise, but your brain focuses on your friend's voice and isolates it because that's what you want to hear. Similarly, the model learns an efficient way to represent the original data so that it can reconstruct it without the anomalous interference noise.

This can also be a case where an image has been altered in some form, such as by having perturbations, loss of detail, fog, etc. The model learns an accurate representation of the original image and outputs a reconstruction without any of the anomalous elements in the image.

Novelty Detection

Novelty detection is very similar to outlier detection. In this case, a novelty is a data point outside of the training set, the data set the model was exposed to, that was shown

to the model to determine if it is an anomaly or not. The key difference between novelty detection and outlier detection is that in outlier detection, the job of the model is to determine what is an anomaly within the training data set. In novelty detection, the model learns what is a normal data point and what isn't, and tries to classify anomalies in a new data set that it has never seen before.

The Three Styles of Anomaly Detection

It is important to note that there are three overarching “styles” of anomaly detection. They are

- **Supervised anomaly detection**
- **Semi-supervised anomaly detection**
- **Unsupervised anomaly detection**

Supervised anomaly detection is a technique in which the training data has labels for both anomalies and for normal data points. Basically, you tell the model during the training process if a data point is an anomaly or not. Unfortunately, this isn't the most practical method of training, especially because the entire data set needs to be processed and each data point needs to be labeled. Since supervised anomaly detection is basically a type of binary classification task, meaning the job of the model is to categorize data under one of two labels, any classification model can be used for the task, though not every model can attain a high level of performance. An example of this can be seen in Chapter 7 with the temporal convolutional network.

Semi-supervised anomaly detection involves partially labeling the training data set. In the context of anomaly detection, this can be a case where only the normal data is labeled. Ideally, the model will learn what normal data points look like, so that the model can flag anomalous data points as anomalies since they differ from normal data points. Examples of models that can use semi-supervised learning for anomaly detection include autoencoders, which you will learn about in Chapter 4.

Unsupervised anomaly detection, as the name implies, involves training the model on unlabeled data. After the training process, the model is expected to know what data points are normal and what points are anomalous within the data set. Isolation forest, a model you will explore in Chapter 2, is one such model that can be used for unsupervised anomaly detection.

Where Is Anomaly Detection Used?

Whether we realize it or not, anomaly detection is being utilized in nearly every facet of our lives today. Pretty much any task involving data collection of any sort could have anomaly detection applied to it. Let's look at some of the most prevalent fields and topics that anomaly detection can be applied in.

Data Breaches

In today's age of big data, where huge volumes of information are stored about users in various companies, information security is vital. Any information breaches must be reported and flagged immediately, but it is hard to do so manually at such a scale. Data leaks can range from simple accidents such as losing a USB stick that contains a company's sensitive information to employees intentionally sending data to an outside party to intrusion attacks that attempt to gain access to the database. You must have heard of some high-profile data leaks, such as the Facebook security breach, the iCloud data breach, and the Google security breach where millions of passwords were leaked. All of those companies operate on an international scale, requiring automation to monitor everything in order to ensure the fastest response time to any breach.

The data breaches might not even need network access. For example, an employee could email an outside party or another employee with connections to rival companies about travel plans to meet up and exchange confidential information. Anomaly detection models can sift through and process employee emails to flag any suspicious employees. The software can pick up key words and process them to understand the context and decide whether or not to flag an employee's email for review.

When employees try to upload data to another connection, the anomaly detection software can pick up on the unusual flow of data while monitoring network traffic and flag the employee. An important part of an employee's regular work day would be to pull and push to a code repository, so one might expect regular spikes in data transfer in these cases. However, the software takes into account lots of variables, including who the sender is, who the recipient is, how the data is being sent (in erratic intervals, all at once, or spread out over time). In either case, something won't add up, which the software will pick up and then it will flag the employee.

The key benefit to using anomaly detection in the workspace is how easy it is to scale up. These models can be used for small companies as well as large-scale international companies.

Identity Theft

Identity theft is another common problem in today's society. Thanks to the development of online services allowing for ease of access when purchasing items, the volume of credit card transactions that take place every day has grown immensely. However, this development also makes it easier to steal credit card information or bank account information, allowing the criminals to purchase anything they want if the card isn't deactivated or if the account isn't secured again. Because of the huge volume of transactions, it can get hard to monitor everything. However, this is where anomaly detection can step in and help, since it is highly scalable and can help detect fraud transactions the moment the request is sent.

As you saw earlier, context matters. If a transaction is made, the software will take into account the card holder's previous history to determine if it should be flagged or not. Obviously, a high value purchase made suddenly would raise alarms immediately, but what if the criminals were smart enough to realize that and just make a series of purchases over time that won't put a noticeable hole in the card holder's account? Again, depending on the context, the software would pick up on these transactions and flag them again.

For example, let's say that someone's grandmother was recently introduced to Amazon and to the concept of buying things online. One day, unfortunately, she stumbles upon an Amazon lookalike and enters her credit card information. On the other side, some criminal takes it and starts buying random things, but not all at once so as not to raise suspicion—or so he thought. The identify theft insurance company starts noticing some recent purchases of batteries, hard drives, flash drives, and other electronic items. While these purchases might not be that expensive, they certainly stand out when all the purchases made by the grandmother up until now consisted of groceries, pet food, and various decoration items. Based on this previous history, the detection software would flag the new purchases and the grandmother would be contacted to verify these purchases. These transactions can even be flagged as soon as an attempt to purchase is made. In this case, either the location or the transactions themselves would raise alarms and stop the transaction from being successful.

Manufacturing

You explored a use case of anomaly detection in manufacturing. Manufacturing plants usually have a certain level of quality that they must ensure that their products meet before shipping them out. When factories are configured to produce massive quantities

of output at a near constant rate, it becomes necessary to automate the process of checking the quality of various samples. Similar to the screw example, manufacturing plants in real life might test to uphold the quality of various metal parts, tools, engines, food, clothes, etc.

Networking

Perhaps one of the most important use cases that anomaly detection has is in networking. The internet is host to a vast array of various websites that are located all around the world. Unfortunately, due to the ease of access to the Internet, various individuals can access the Internet with nefarious purposes. Similar to the data leaks that were discussed earlier in the context of protecting company data, hackers can launch attacks on other websites as well to leak their information.

One such example is hackers attempting to leak government secrets through a network attack. With such sensitive information as well as the high volumes of expected attacks every day, automation is a necessary tool to help cybersecurity professionals deal with the attacks and preserve state secrets. On a smaller scale, hackers might attempt to breach individual cloud networks or a local area network and try to leak data. Even in smaller cases like this, anomaly detection can help detect network intrusion attacks as they happen and notify the proper officials. An example data set for network intrusion anomaly detection is the KDD Cup 1999 data set. This data set contains a large amount of entries that detail various types of network intrusion attacks as well as a detailed list of variables for each attack that can help a model identify each type of attack.

Medicine

Moving on from networking, anomaly detection has a massive role to play in the field of medicine. For example, models can detect subtle irregularities in a patient's heartbeat in order to classify diseases, or they can measure brainwave activity to help doctors diagnose certain conditions. Beyond that, they can help analyze raw diagnostic data for a patient's organ and process it in order to quickly diagnose any possible problems within the patient, similarly to the thyroid example discussed earlier.

Anomaly detection can even be used in medical imagery to determine if a given image contains anomalous objects or not. For example, if a model was only exposed to MRI imagery of normal bones and was shown an image of a broken bone, it would flag

the new image as an anomaly. Similarly, anomaly detection can even be extended to tumor detection, allowing for the model to analyze every image in a full body MRI scan and look for the presence of abnormal growth or patterns.

Video Surveillance

Anomaly detection also has uses in video surveillance, where anomaly detection software can monitor video feeds and help flag any videos that capture anomalous action. While this might seem dystopian, it can certainly help catch criminals or maintain public safety on busy streets or in cities. For example, this software could identify a mugging in a street at night as an anomalous event and alert authorities who can call in police officers. Additionally, it can detect unusual events at crossroads such as an accident or some unusual obstruction and immediately call attention to the footage.

Summary

Generally, anomaly detection is utilized heavily in medicine, finance, cybersecurity, banking, networking, transportation, and manufacturing, but it is not just limited to those fields. For nearly every case imaginable involving data collection, anomaly detection can be put to use to help users automate the process of detecting anomalies and possibly removing them. Many fields in science can utilize anomaly detection because of the large volume of raw data collection that goes on. Anomalies that would interfere with the interpretation of results or otherwise introduce some sort of bias into the data could be detected and removed, provided that the anomalies are caused by systematic or random errors.

In this chapter, we discussed what anomalies are and why detecting anomalies can be very important to the data processing we have at our organizations.

In the next chapter, we will look at traditional statistical and machine learning algorithms for anomaly detection.

CHAPTER 2

Traditional Methods of Anomaly Detection

In this chapter, you will learn about traditional methods of anomaly detection. You will also learn how various statistical methods and machine learning algorithms work and how they can be used to detect anomalies and how you can implement anomaly detection using several algorithms.

In a nutshell, the following topics will be covered throughout this chapter:

- A data science review
- The three styles of anomaly detection
- The isolation forest
- One-class support vector machine (OC-SVM)

Data Science Review

It is important to understand some basic data science concepts in order for you to evaluate how well your model performs and to compare its performance with other models.

First of all, the goal in anomaly detection is to determine whether or not a given point is an anomaly or not. Essentially, you are labeling a data point $\mathbf{x}$ with a class $\mathbf{y}$. Assume that in some context, you are trying to classify whether or not an animal tests positive (meaning yes) for some disease. If the animal is diseased and it tests positive, this case is a **true positive**. If the animal is healthy and the test shows negative (meaning it doesn't have the disease), then it's called a **true negative**. However, there are cases

where the test can fail. If the animal is healthy but the test says positive, this case is a **false positive**. If the animal is diseased but the test shows negative, this case is a **false negative**.

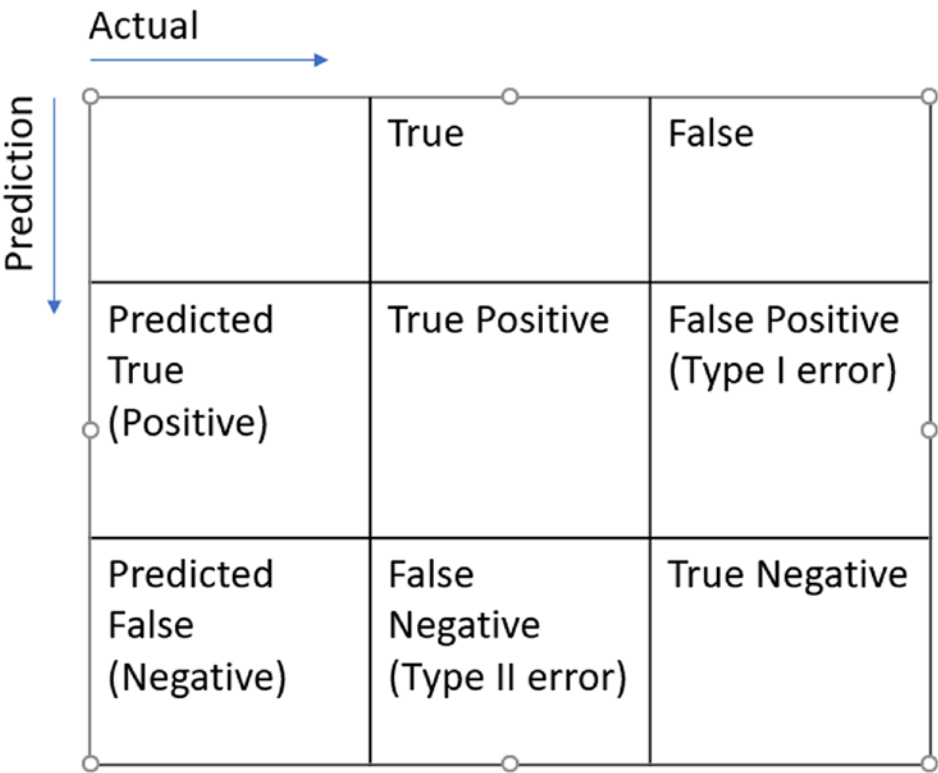
In statistics, there are similar terms to false positive and false negative: **type I error** and **type II error**. These errors are used in hypothesis testing where you have a null hypothesis (which usually says that there is no relation between two observed phenomena), and an alternate hypothesis (which aims to disprove the null hypothesis, meaning there is a statistically significant relation between the two observations).

A **type I error** is when the null hypothesis turns out to be true, but you reject it anyways in favor of the alternate hypothesis. In other words, a false positive, since you reject what turns out to be true to accept something that is false. A **type II error** is when the null hypothesis is accepted to be true (meaning you don't reject the null hypothesis), but it turns out the null hypothesis is false, and that the alternate hypothesis is true. This is a false negative, since you accept what is false, but reject what is true.

For the context of the following definitions, assume that the condition is what you're trying to prove. It could be something as simple as "this is animal sick." The condition of the animal is either sick or healthy, and you're trying to predict if it is sick or healthy. Here are some definitions:

- **True positive:** When the condition is true, and the prediction is also true
- **True negative:** When the condition is false, and the prediction is also false
- **False positive:** When the condition is false, but the prediction is true
- **False negative:** When the condition is true, but the prediction is false

Putting them together, you can form what is called a **confusion matrix** (Figure 2-1). One thing to note is that in the case of anomaly detection, you only need a 2x2 confusion matrix since data points are either anomalies or they are normal data.



The diagram shows a 2x2 confusion matrix. A horizontal blue arrow labeled 'Actual' points to the right above the top row. A vertical blue arrow labeled 'Prediction' points downwards to the left of the first column. The matrix cells are labeled as follows:

	Actual True	Actual False
Predicted True (Positive)	True Positive	False Positive (Type I error)
Predicted False (Negative)	False Negative (Type II error)	True Negative

Figure 2-1. *Confusion matrix*

From the values in each of the four squares, you can derive values for **accuracy**, **precision**, and **recall** to gain a better understanding of how your model performs. Here’s the confusion matrix with all the formulas (Figure 2-2):

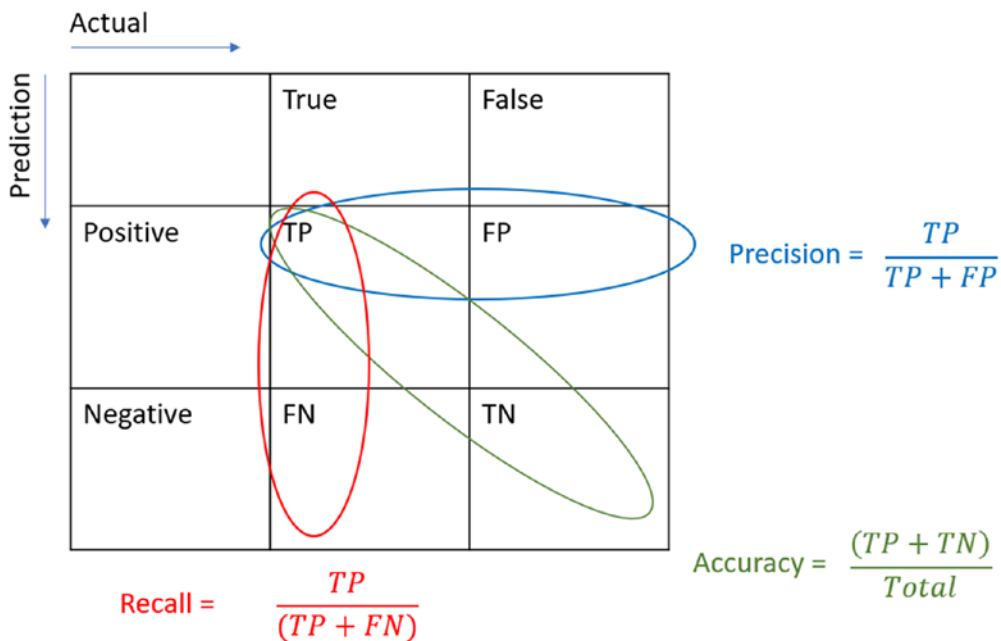


Figure 2-2. Precision, Accuracy and Recall

- **Precision** is a measurement that describes how many of your true predictions actually turned out to be true. In other words, for all of your true predictions, how many did the model get right?
- **Accuracy** is a measurement that describes how many predictions you got right over the entire data set. In other words, for the entire data set, how many did the model correctly predict were positive and negative?
- **Recall** is a measurement that describes how many you predicted true for all data points that were actually true. In other words, for all of the true data points in the data set, how many of them did the model predict correctly?

From here, you can derive more values.

F1 Score is the harmonic mean of precision and recall. It's a metric that can tell us how accurate the model is, since it takes into account both how well the model makes true predictions that are actually true, and how many of the total true predictions that the model correctly predicted.

$$\text{F1 Score} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

The **true positive rate (TPR) = recall = sensitivity**. The same as recall, the TPR tells us how many of the data points that are actually true were predicted as true by the model.

$$\text{The false positive rate (FPR)} = (1 - \text{specificity}) = \frac{FP}{FP + TN}$$

The FPR tells us how many of the data points that are actually false were predicted to be positive by the model. The formula is similar to recall, but instead of the proportion of true positives to all of the true data points, it's the proportion of false positives to all of the false data points.

$$\text{Specificity} = 1 - \text{FPR} = \frac{TN}{TN + FP}$$

Specificity is very similar to recall in that it tells us how many of the data points that are actually false were predicted as false by the model.

We can use the TPR and the FPR to form a graph known as a **receiver operating characteristic** curve, or **ROC** curve. From the **area under the curve**, or **AUC** (you may see this called **area under the curve of the receiver operating characteristic**, or **AUROC**), a data point, meaning the probability of the model to have a true positive or true negative case. This curve can also be called an **AUCROC** curve.

ROC curve with AUC = 1.0 (Figure 2-3).

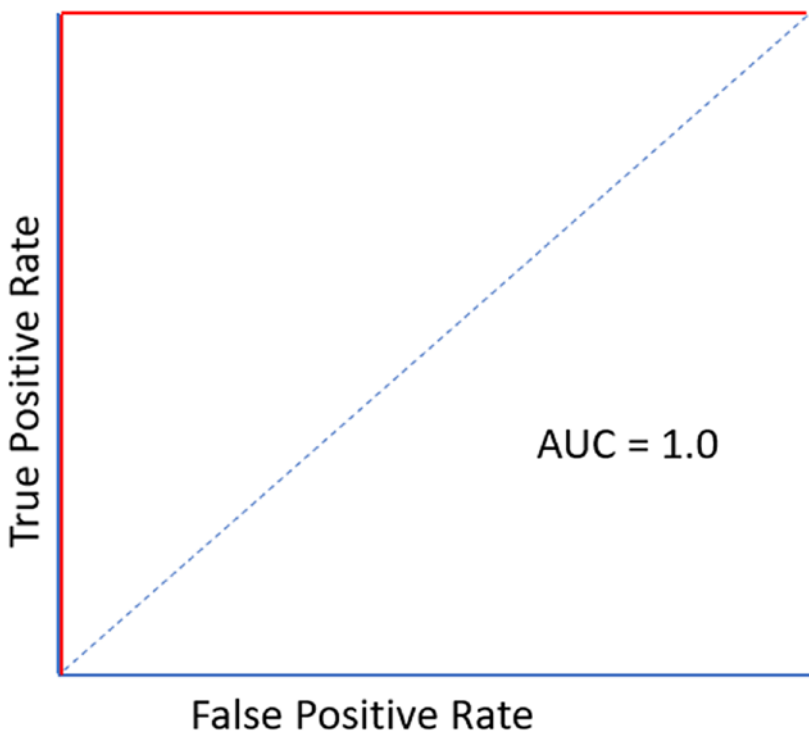


Figure 2-3. ROC curve with $AUC = 1.0$

This is the most ideal AUC curve. However, it is nearly impossible to attain, so a goal of $AUC > 0.95$ is most desirable. The closer we can get the model to attaining a value of 1.0 for the AUC, the more the probability of the model to predict a true positive or true negative case. The AUC value in the graph above indicates that this probability is 1.0, meaning it predicts it correctly 100% of the time. However, an extremely high AUC value of say 0.99999 could indicate that the model is **overfitting**, meaning it's getting really good at predicting labels for this particular data set. You will explore this concept a bit further in the context of support vector machines, but you want to avoid overfitting as much as possible so that the model can perform well even when introduced to new data that includes unexpected variations.

It is important to mention that although the AUC can be 0.99, for example, it is not guaranteed that the model will continue to perform at that high of a level outside of the **training data set** (the data used to train the model so that it can learn to classify anomalies and normal data). This is because in the real world, there is the factor of unpredictability that even has humans confused at times. The world would be a simpler place if data is black and white, so to speak, but more often than not, there is a huge

gray area (are we sure that point is X and not Y? Is this really an anomaly or just a really weird case of a normal point?). For deep learning models, it is important that they keep achieving high AUC scores when exposed to new data that includes plenty of variation. Basically, it's a reasonable assumption to expect a slight drop in performance when exposing your model to new data outside of your training set.

The goal with training models is to avoid overfitting and to keep the AUC as high as possible. If the AUC turns out to be 0.99999 even after being exposed to an extremely large sample of new data that includes a lot of variety, that means the model is basically about as ideal of a model we can get and has far surpassed human performance, which is impossible for the time being.

ROC curve with AUC = 0.75 (Figure 2-4)

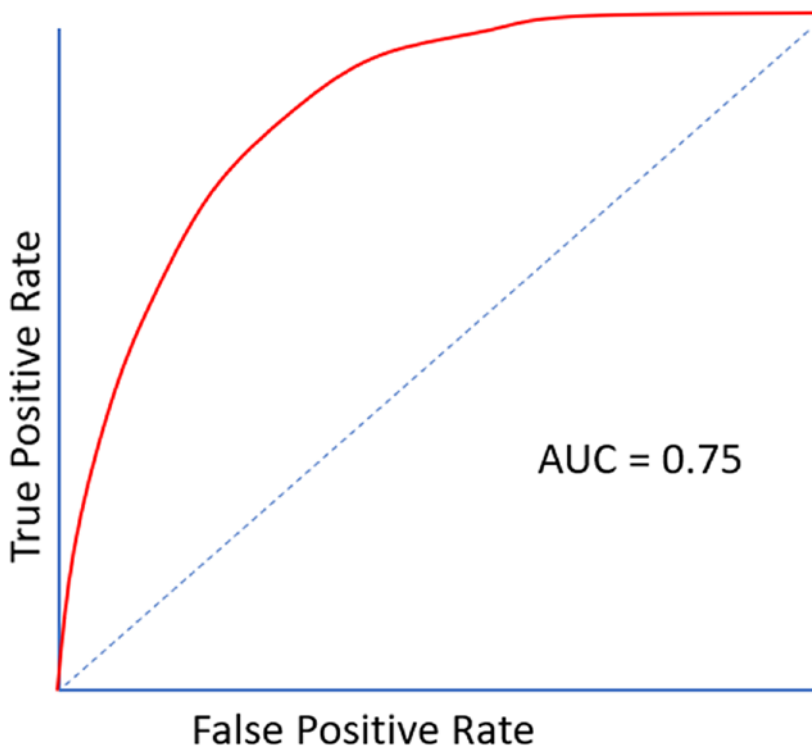


Figure 2-4. ROC curve with AUC = 0.75

The value for the AUC indicates that the model correctly predicts labels for data points only 75% of the time. It’s not bad, but it’s not good, so there’s clearly room to improve.

ROC curve with AUC = 0.5 (Figure 2-5)

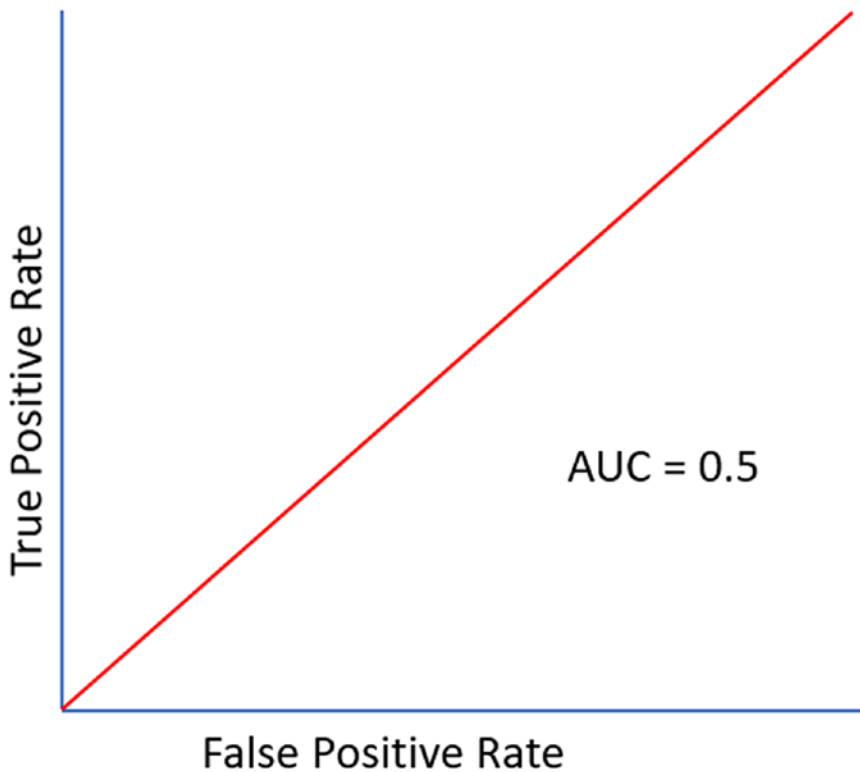


Figure 2-5. ROC curve with AUC = 0.5

The value for the AUC indicates that the model only has a 50% chance, or a probability of 0.5, to predict the correct label. This is about the worst AUC value you can get, since it means the model cannot distinguish between the positive and negative classes.

ROC curve with AUC = 0.25 (Figure 2-6)

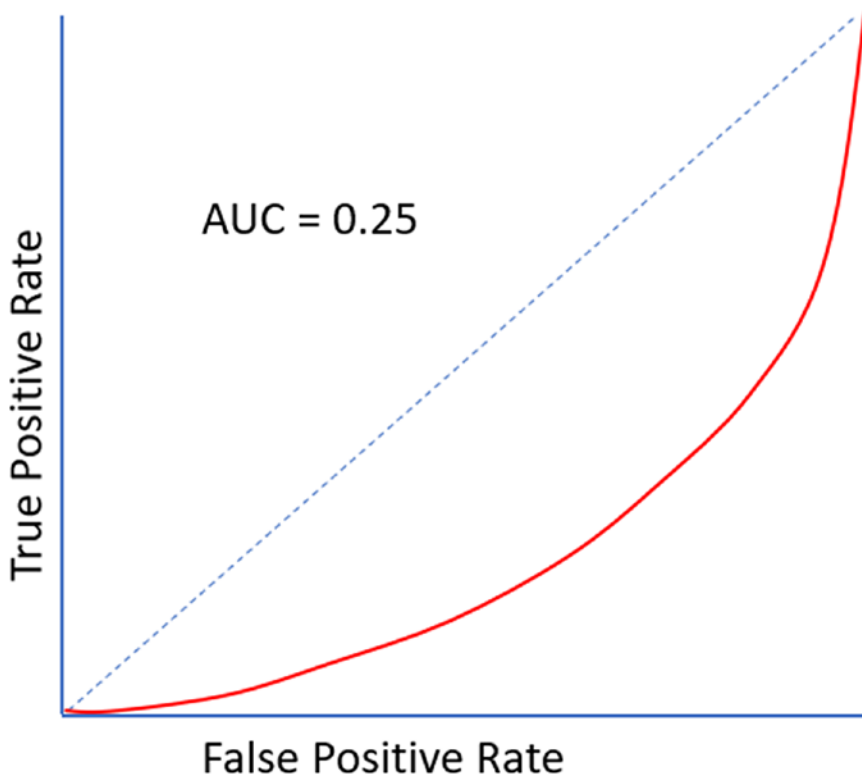


Figure 2-6. ROC curve with $AUC = 0.25$

In this case, the model only has a probability of 0.25 to predict the right label, but this just means that it has a 0.75 probability of predicting the incorrect label. In the case that the AUC is 0, this means that the model is perfect at predicting the wrong label, meaning the labels are switched. If the AUC is < 0.5 , this means the model gets better at predicting incorrectly as the AUC approaches 0.0. It's the perfectly opposite case of when the AUC is > 0.5 , where the model gets better at predicting correctly as the AUC approaches 1.0.

In any case, you want the AUC to be > 0.5 , and at least greater than 0.9 and ideally greater than 0.95.

Isolation Forest

An isolation forest is a collection of individual tree structures that recursively partition the data set. In each iteration of the process, a random feature is selected, and the data is split based on a randomly chosen value between the minimum and maximum of the chosen feature. This is repeated until the entire data set is partitioned to form an individual tree in the forest. Anomalies generally form much shorter paths from the root than normal data points since they are much more easily isolated. You can find the anomaly score by using a function of the data point involving the average path length.

Applying an isolation forest to an unlabeled data set in order to catch anomalies is an example of **unsupervised anomaly detection**.

Mutant Fish

To better understand what an isolation forest does, let's look at an imaginary scenario. At a particularly large lake, an irresponsible fish breeder has released a mutant species of fish that looks eerily similar to the native species, but are on average bigger than the native species. Additionally, the proportion of the length of its tail fin to the length of its body is larger than the native species. All in all, there are three features you can use to distinguish the invasive, mutant species from the native species.

Here's a visual example detailing the differences of an average specimen of both species. You can see the **native species** in Figure 2-7.



Figure 2-7. *This is an example of the native species at this lake*

You can see the **invasive species** in Figure 2-8.

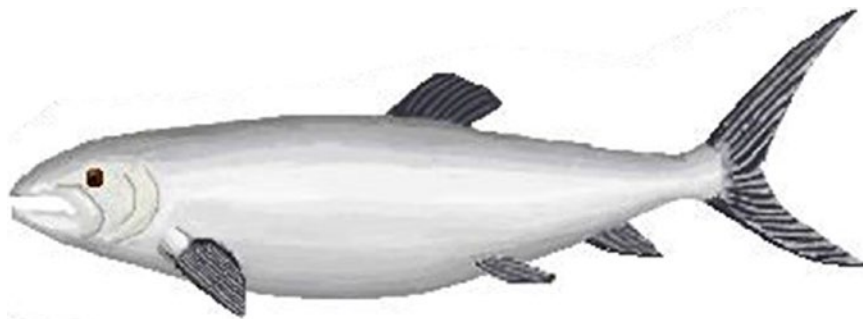


Figure 2-8. *This is an example of the new, mutant species that has been released into the lake*

The invasive species is larger, has a bigger circumference, and has a longer tailfin on average (compare Figure 2-7 to Figure 2-8). However, the problem is that while the average specimen of each species has some noticeable distinctions between them, there is plenty of overlap between the two species where some of the native species grow large, some of the mutant species are just smaller, both have varying tail fin sizes, etc. so the differences might not always be as clear-cut.

To find out the extent of this infiltration, a large group of fishermen have been assembled and presented with the task of identifying the species of each fish in a catch of 1,000 fish. In this case, assume that each fisherman will randomly profile each fish to determine whether it is a member of the native species or not.

Now onto the evaluations. Each fisherman first picks a random feature to judge the samples on: the length of the fish, the circumference of the fish, or the proportion of its tail fin to its overall length. Then, the fisherman picks a random value between the known minimum and maximum values of the corresponding measurement for the native species and splits all the fish accordingly (all fish with the relevant measurement equal to or bigger than the picked value go right, and everything else goes left, for example). The fisherman repeats the entire process over and over again until every single fish has been partitioned and a “tree” of fish has been created.

In this case, each individual fisherman represents a tree in the isolation forest, and the resulting trees of the entire group of fishermen represent an isolation forest. Now, given a random fish in the entire catch, you can get an anomaly score to see how many of the fisherman found that this fish is anomalous. Based on the threshold you pick for the anomaly score, you can label certain fish as the invasive species and the others as the native species.

However, the problem is that this is not a perfect system; there will be some invasive fish that pass off as native fish, and some native fish that pass off as invasive species. These cases represent false positives and false negatives.

Anomaly Detection with Isolation Forest

Now that you understand more about how an isolation forest works, you can move on to applying it to a data set. Before you start, it is important to note that an isolation forest performs well on high-dimensional data. For the invasive fish example, you had three features to work with: fish length, circumference, and proportion of tail fin length to overall length. In this next example, you will have 42 features per data entry.

You will use the KDDCUP 1999 data set, which contains an extensive amount of data representing a wide variety of intrusion attacks. In particular, you will focus on all data entries that involve an HTTP attack. The data set can be found at <http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html>. After opening the link, you should see something like Figure 2-9.

KDD Cup 1999 Data

Abstract

This is the data set used for The Third International Knowledge Discovery and Data Mining Tools detector, a predictive model capable of distinguishing between ``bad" connections, called intrusion:

Information files:

- [task description](#). This is the original task decription given to competition participants.

Data files:

- [kddcup.names](#) A list of features.
- [kddcup.data.gz](#) The full data set (18M; 743M Uncompressed)
- [kddcup.data 10 percent.gz](#) A 10% subset. (2.1M; 75M Uncompressed)
- [kddcup.newtestdata 10 percent unlabeled.gz](#) (1.4M; 45M Uncompressed)
- [kddcup.testdata.unlabeled.gz](#) (11.2M; 430M Uncompressed)
- [kddcup.testdata.unlabeled 10 percent.gz](#) (1.4M;45M Uncompressed)
- [corrected.gz](#) Test data with corrected labels.
- [training attack types](#) A list of intrusion types.
- [typo-correction.txt](#) A brief note on a typo in the data set that has been corrected (6/26/07)

[The UCI KDD Archive](#)
[Information and Computer Science](#)
[University of California, Irvine](#)
 Irvine, CA 92697-3425
 Last modified: October 28, 1999

Figure 2-9. *This is what you should see when you open the link*

Download the kddcup.data.gz file and extract it.

There shouldn't be any issues with version mismatch and code functionality, but just in case, the exact Python 3 packages used in this example are as follows:

- numpy 1.15.3
- pandas 0.23.4
- scikit-learn 0.19.1
- matplotlib 2.2.2

First, import all the necessary modules that your code calls upon (Figure 2-10).

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.ensemble import IsolationForest
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

%matplotlib inline
```

Figure 2-10. *Importing numpy, pandas, matplotlib.pyplot, and sklearn modules*

The module **numpy** is a dependency of many of the other modules since it allows them to perform high levels of computation. **Pandas** is a module that allows us to read data files of various formats in order to store them as data frame objects, and it is a popular framework for data science in general. These data frames hold data entries in a similar fashion to arrays and can be thought of as a table of values. **Matplotlib** is a Python library that allows us to customize and plot data. Finally, **scikit-learn** is a package that allows us to apply various machine learning models to data sets as well as provide tools for data analysis.

`%matplotlib inline` allows for graphs to be displayed below the cell and to be saved alongside the notebook.

Next, define the columns and load the data frame (Figure 2-11).

```

columns = ["duration", "protocol_type", "service", "flag", "src_bytes",
"dst_bytes", "land", "wrong_fragment", "urgent",

        "hot", "num_failed_logins", "logged_in", "num_compromised",
"root_shell", "su_attempted", "num_root",

        "num_file_creations", "num_shells", "num_access_files",
"num_outbound_cmds", "is_host_login",

        "is_guest_login", "count", "srv_count", "serror_rate",
"srv_serror_rate", "rerror_rate", "srv_rerror_rate",

        "same_srv_rate", "diff_srv_rate", "srv_diff_host_rate",
"dst_host_count", "dst_host_srv_count",

        "dst_host_same_srv_rate", "dst_host_diff_srv_rate",
"dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate",

        "dst_host_serror_rate", "dst_host_srv_serror_rate",
"dst_host_rerror_rate", "dst_host_srv_rerror_rate", "label"]

df = pd.read_csv("datasets/kdd_cup_1999/kddcup.data/kddcup.data.corrected",
sep=",", names=columns, index_col=None)

```

Figure 2-11. You define all of the columns and save the data set as a variable named *df*

Each data entry is massive, with 42 columns of data per entry. The exact name doesn't matter, but it's important to have "service" and "label" stay the same. The entire list of columns names is as follows:

- duration
- protocol_type
- service
- flag
- src_bytes
- dst_bytes
- land
- wrong_fragment
- urgent
- hot

- num_failed_logins
- logged_in
- num_compromised
- root_shell
- su_attempted
- num_root
- num_file_creations
- num_shells
- num_access_files
- num_outbound_cmds
- is_host_login
- is_guest_login
- count
- srv_count
- serror_rate
- srv_serror_rate
- rerror_rate
- srv_rerror_rate
- same_srv_rate
- diff_srv_rate
- srv_diff_host_rate
- dst_host_count
- dst_host_srv_count
- dst_host_same_srv_rate
- dst_host_diff_srv_rate
- dst_host_same_src_port_rate

- `dst_host_srv_diff_host_rate`
- `dst_host_serror_rate`
- `dst_host_srv_serror_rate`
- `dst_host_rerror_rate`
- `dst_host_srv_rerror_rate`
- `label`

To get the dimensions of the table, or **shape**, as it's referred to in pandas, do
`df.shape`

or if you're not in Jupyter, do

`print(df.shape)`

In Jupyter, you should see something like Figure 2-12 after running the code.

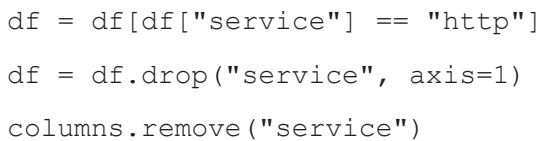


```
In [87]: 1 df.shape
Out[87]: (4898431, 42)
```

Figure 2-12. The output is a tuple that describes the dimensions of the data frame

As you can see, this is a massive dataset.

Next, filter out the entire data frame to only include data entries that involve an HTTP attack, and drop the service column (Figure 2-13).



```
df = df[df["service"] == "http"]
df = df.drop("service", axis=1)
columns.remove("service")
```

Figure 2-13. Filtering `df` to only have HTTP attacks and removing the service column from `df`

Just to make sure, check the shape of `df` again (Figure 2-14).

```
In [91]: 1 df.shape
Out[91]: (623091, 41)
```

Figure 2-14. *The dimensionality of the filtered df*

The number of rows has been drastically reduced, and the column count went down by one because you removed the service column since you don't actually need it anymore.

Let's check all the possible labels and the number of counts for each label, just to get a feel of the data distribution.

Run the following:

```
df["label"].value_counts()
```

or

```
print(df["label"].value_counts())
```

You should see something like Figure 2-15.

```
In [93]: 1 df["label"].value_counts()
Out[93]: normal.      619046
         back.        2203
         neptune.     1801
         portsweep.    16
         ipsweep.     13
         satan.        7
         phf.          4
         nmap.         1
         Name: label, dtype: int64
```

Figure 2-15. *The unique labels in df along with the number of instances of data points in df with that specific label*

The vast majority of the data set is comprised of normal data entries, with around 0.649% of data entries for all HTTP attacks comprising actual intrusion attacks.

Additionally, some of the columns have categorical data values, meaning the model will have trouble training on them. To bypass this issue, you use a built-in feature of scikit-learn called a **label encoder**.

Figure 2-16 shows what you currently see if you run `df.head(5)`, meaning you want five entries to display.

```
In [97]: 1 df.head(5)
```

```
Out[97]:
```

	duration	protocol_type	flag	src_bytes	dst_bytes
0	0	tcp	SF	215	45076
1	0	tcp	SF	162	4528
2	0	tcp	SF	236	1228
3	0	tcp	SF	233	2032
4	0	tcp	SF	239	486

Figure 2-16. A line of code to display the top five entries in the table. In this case, the image has been cropped to show the first few columns

You can also run `print(df.head(5))`, but it prints in a text format (Figure 2-17).

```
In [98]: 1 print(df.head(5))
```

```

duration protocol_type flag  src_bytes  dst_bytes  land  wrong_fragment  \
0         0          tcp   SF        215     45076    0         0
1         0          tcp   SF        162     4528    0         0
2         0          tcp   SF        236     1228    0         0
3         0          tcp   SF        233     2032    0         0
4         0          tcp   SF        239      486    0         0

urgent  hot  num_failed_logins  logged_in  num_compromised  root_shell  \
0       0   0                0          1                0         0
1       0   0                0          1                0         0
2       0   0                0          1                0         0
3       0   0                0          1                0         0
4       0   0                0          1                0         0

```

Figure 2-17. The same function as in Figure 2-16, but in text format

To resolve this issue, the **label encoder** takes the unique (meaning one entry per categorical value instead of multiple) list of categorical values and assigns a number representing each of them. If you had an array like

```
[ "John", "Bob", "Robert"],
```

the label encoder would create a numerical representation like

```
[0, 1, 2],
```

where 0 represents "John", 1 represents "Bob", and 2 represents "Robert."

Now do the same with the labels in your data frame.

Run the code in Figure 2-18.

```
for col in df.columns:
    if df[col].dtype == "object":
        encoded = LabelEncoder()
        encoded.fit(df[col])
        df[col] = encoded.transform(df[col])
```

Figure 2-18. Applying the label encoder to the columns with data values that are strings

`encoded.fit(df[col])` gives the label encoder all of the data in the column from which it extracts the unique categorical values from. When you run

```
df[col] = encoded.transform(df[col])
```

you are assigning the encoded representation of each categorical value to `df[col]`.

Let's check the data frame now (Figure 2-19).

```
In [101]: 1 df.head(5)
```

```
Out[101]:
```

	duration	protocol_type	flag	src_bytes	dst_bytes	land
0	0	0	9	215	45076	0
1	0	0	9	162	4528	0
2	0	0	9	236	1228	0
3	0	0	9	233	2032	0
4	0	0	9	239	486	0

Figure 2-19. Looking at the first five entries of `df` after applying the label encoder

Good, all the categorical values have been replaced with numerical equivalents.

Now run the code in Figure 2-20.

```

for f in range(0, 3):
    df = df.iloc[np.random.permutation(len(df))]

df2 = df[:500000]
labels = df2["label"]
df_validate = df[500000:]

x_train, x_test, y_train, y_test = train_test_split(df2, labels,
test_size = 0.2, random_state = 42)

x_val, y_val = df_validate, df_validate["label"]

```

Figure 2-20. *Shuffling the values in df and creating your training, testing, and validation data sets*

With

```
df = df.iloc[np.random.permutation(len(df))]
```

you are randomly shuffling all the entries in the data set to avoid the problem of abnormal entries pooling in any one region of the data set.

With

```
df2 = df[:500000]
```

you are assigning the first 500,000 entries of df to a variable df2.

In the next line of code, `labels = df2["label"]`, you assign the label column to the variable labels. Next, you assign the rest of the data frame to a variable named `df_validate` to create the validation data set with `df_validate = df[500000:]`.

To split your data into the **training set** and **testing set**, you can use a built-in scikit-learn function called `train_test_split`, as detailed below:

```
x_train, x_test, y_train, y_test = train_test_split(df2, labels,
test_size = 0.2, random_state = 42)
```

The parameters are as follows: x, y, test_size, and random_state. Note that x and y are supposed to be the training data and training labels, respectively, with test_size indicating the percentage of the data set to be used as test data. random_state is a

number used to initialize the random number generator that determines what data entries are chosen for the training data set and for the test data set.

Finally, you delegate the rest of the data to the **validation set**. To define the terms again:

- **Training data** is the data that the model trains and learns on. For an isolation forest, this set is what the model partitions on. For neural networks, this set is what the model adjusts its weights on.
- **Testing data** is the data that is used to test the model's performance. The `train_test_split()` function basically splits the data into a portion used to train on and a portion used to test the model's performance on.
- **Validation data** is used during training to gauge how the model's training is going. It basically helps ensure that as the model gets better at performing the task on the training data, it also gets better at performing the same task over new, but similar data. This way, the model doesn't only get really good at performing the task on the training data, but can perform similarly on new data as well. In other words, you want to avoid **overfitting**, a situation where the model performs very well on a particular data set, which can be the training data set, yet the performance noticeably drops when new data is presented. A slight drop in performance is to be expected when the model is exposed to new variations in the data, but in this case, it is more pronounced.

In this example, you don't use the validation set or testing set during training, but this will come into play later on when you are training neural networks. Instead, you use them to evaluate the performance of the model.

Let's take a look at the shapes of your new variables by running the code in Figure 2-21.

```
In [140]: 1 print("Shapes:\nx_train:%s\ny_train:%s\n" % (x_train.shape, y_train.shape))
          2 print("x_test:%s\ny_test:%s\n" % (x_test.shape, y_test.shape))
          3 print("x_val:%s\ny_val:%s\n" % (x_val.shape, y_val.shape))
          4
          5

Shapes:
x_train: (400000, 41)
y_train: (400000,)

x_test: (100000, 41)
y_test: (100000,)

x_val: (123091, 41)
y_val: (123091,)
```

Figure 2-21. Getting the shapes of the training, testing, and validation data sets

To build your isolation forest model, run the following:

```
isolation_forest = IsolationForest(n_estimators=100, max_samples=256,
contamination=0.1, random_state=42)
```

Here's an explanation of the parameters:

- **n_estimators** is the number of trees to use in the forest. The default is 100.
- **max_samples** is the maximum number of data points that the tree should build on. The default is whatever is smaller: 256 or the number of samples in the data set.
- **contamination** is an estimate of the percentage of the entire data set that should be considered an anomaly/outlier. It is 0.1 by default.
- **random_state** is the number it will initialize the random number generator with to use during the training process. An isolation forest utilizes the random number generator quite extensively during the training process.

Now, let's train your isolation forest model by running

```
isolation_forest.fit(x_train)
```

This process will take some time, so get up and stretch for a bit!

Once it's finished, you can go about calculating the anomaly scores. Let's create a histogram of the anomaly scores when tested on the validation set.

Run the code in [Figure 2-22](#).


```

anomaly_scores = isolation_forest.decision_function(x_val)
plt.figure(figsize=(15, 10))
plt.hist(anomaly_scores, bins=100)
plt.xlabel('Average Path Lengths', fontsize=14)
plt.ylabel('Number of Data Points', fontsize=14)
plt.show()

```

Figure 2-22. Getting the anomaly scores from the trained isolation forest model and plotting a histogram

You should see a graph that looks like Figure 2-23.

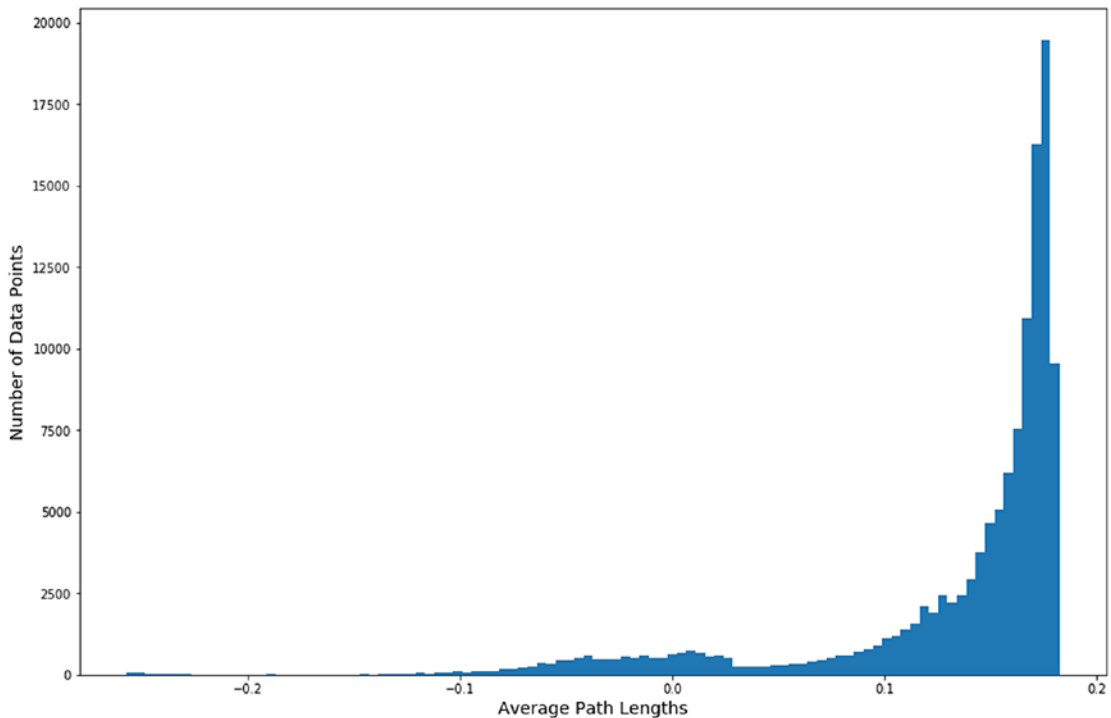


Figure 2-23. A histogram plotting the average path lengths for the data points. It helps you determine what is an anomaly by using the shortest set of path lengths, since that indicates that the model was able to easily isolate those points

A quick note: `plt.show()` is not necessary on Jupyter if you have `%matplotlib inline`, but if you are using anything else, this should open up a new window with the graph.

Let's calculate the **AUC** to see how well the model did. Looking at the graph, there appears to be a few anomalous data with average path of less than -0.15. You expect there to be a few outliers within the normal range of data, so let's pick something more extreme, such as -0.19. Remember that the lesser the path length, the more likely the data is to be anomalous, hence why there's a curve that increases drastically as the graph goes right. Run the code in Figure 2-24.

```
from sklearn.metrics import roc_auc_score

anomalies = anomaly_scores > -0.19
matches = y_val == list(encoded.classes_).index("normal.")
auc = roc_auc_score(anomalies, matches)
print("AUC: {:.2%}".format (auc))
```

Figure 2-24. *Classifying anomalies based on a threshold that you picked from a graph and generating the AUC score from that set of labels for each point*

You should see something like Figure 2-25.

```
In [167]: 1 from sklearn.metrics import roc_auc_score
          2
          3 anomalies = anomaly_scores > -0.19
          4 matches = y_val == list(encoded.classes_).index("normal.")
          5 auc = roc_auc_score(anomalies, matches)
          6 print("AUC: {:.2%}".format (auc))

AUC: 99.81%
```

Figure 2-25. *The generated AUC score after running the code*

That's an impressive score! But could it be the result of overfitting? Let's get the anomaly scores of the test set to find out.

Run the code in Figure 2-26.

```

anomaly_scores_test = isolation_forest.decision_function(x_test)
plt.figure(figsize=(15, 10))
plt.hist(anomaly_scores_test, bins=100)
plt.xlabel('Average Path Lengths', fontsize=14)
plt.ylabel('Number of Data Points', fontsize=14)
plt.show()

```

Figure 2-26. Creating a histogram like in Figure 2-23 for the testing set instead of the validation set

You should get a graph like Figure 2-27.

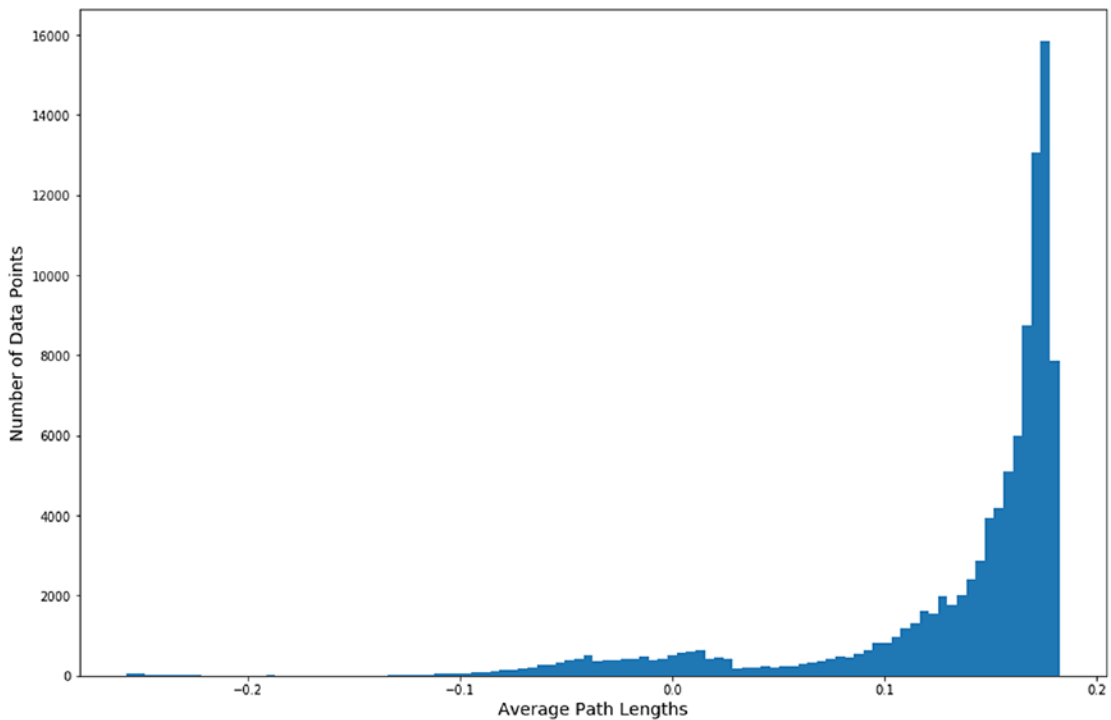


Figure 2-27. A histogram like in Figure 2-23, but for the testing set

There is a similar pattern of what appear to be anomalous data to the left of -0.15. Again, assume that there are expected outliers, and pick any average path length less than -0.19 as the cutoff for anomalies.

Run the code in Figure 2-28.

```

anomalies_test = anomaly_scores_test > -0.19
matches = y_test == list(encoded.classes_).index("normal.")
auc = roc_auc_score(anomalies_test, matches)
print("AUC: {:.2%}".format (auc))

```

Figure 2-28. Applying the code in Figure 2-24 to the test set. In this case, the threshold was the same, but you still picked it based on the histogram

It should look like Figure 2-29.

```

In [163]: 1 anomalies_test = anomaly_scores_test > -0.19
          2 matches = y_test == list(encoded.classes_).index("normal.")
          3 auc = roc_auc_score(anomalies_test, matches)
          4 print("AUC: {:.2%}".format (auc))

AUC: 99.82%

```

Figure 2-29. The generated AUC score for the test set

That’s really good! It seems to perform very well on both the validation data and the test data.

Hopefully by now you will have gained a better understanding of what an isolation forest is and how to apply it. Remember, an isolation forest works well for multi-dimensional data (in this case, you had 41 columns after dropping the service column) and can be used for **unsupervised anomaly detection** when applied in the manner implemented in this section.

One-Class Support Vector Machine

The One-Class SVM is a modified support vector machine model that is well-suited for novelty detection (an example of **semi-supervised anomaly detection**). The idea is that the model trains on normal data and is used to detect anomalies when new data is presented to it. While the OC-SVM might seem best suited to semi-supervised anomaly detection, since training on only one class means it’s still “partially labeled” when considering the entire data set, it can also be used for unsupervised anomaly detection. You will perform semi-supervised anomaly detection on the same KDDCUP 1999 data

set as the isolation forest example. Similar to the isolation forest, the OC-SVM is also good for high-dimensional data. Additionally, the OC-SVM can capture the shape of the data set pretty well, a point that will be elaborated upon below.

To understand how a support vector machine works, first visualize some data on a 2D plane (Figure 2-30).

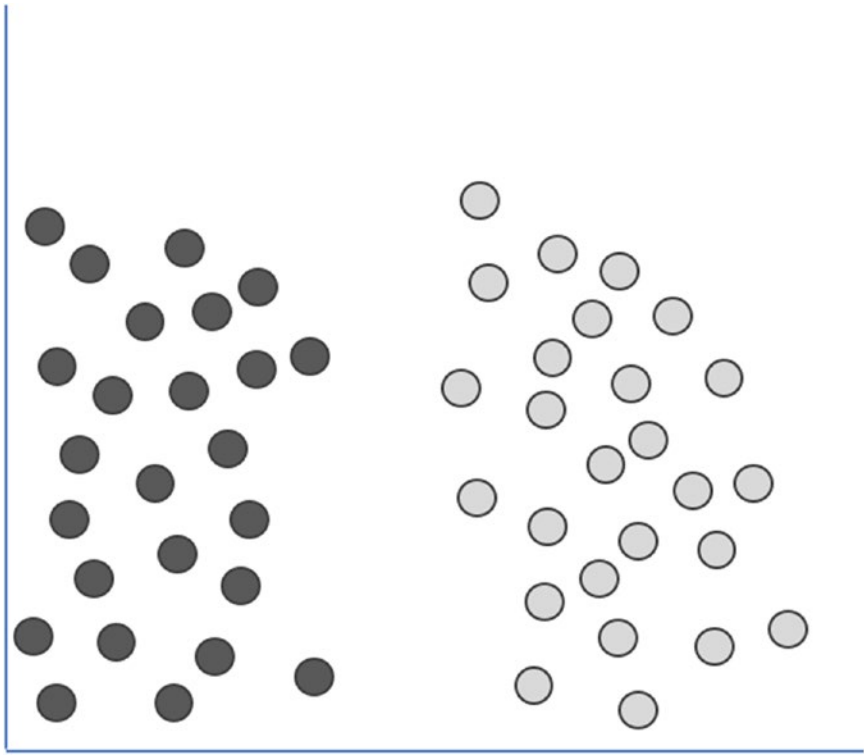


Figure 2-30. *Some points plotted so that they group up in two regions on the graph*

How do you separate the data into two distinct regions using a line? Well, it’s pretty simple (Figure 2-31).

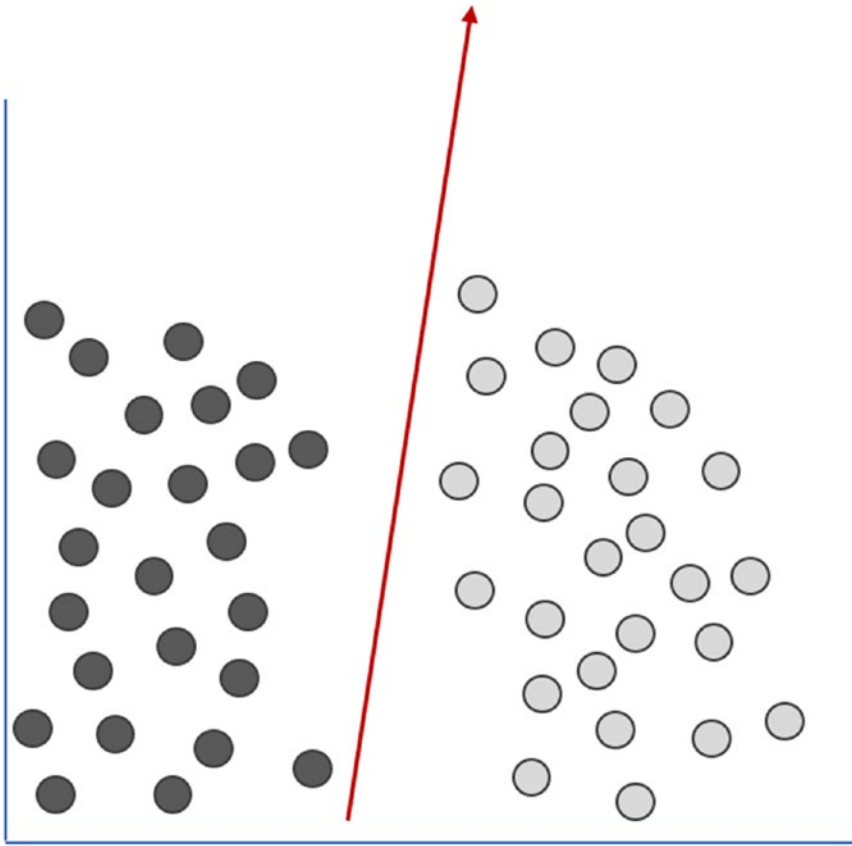


Figure 2-31. *A line that separates the two regions based on the points plotted*

Now you have two regions representing two different labels. However, the problem goes a little bit deeper than that.

The reason the model is called a “support vector machine” is because these “support vectors” actually play a huge role in how the model draws the decision boundary, represented in this case by the line in [Figure 2-32](#).

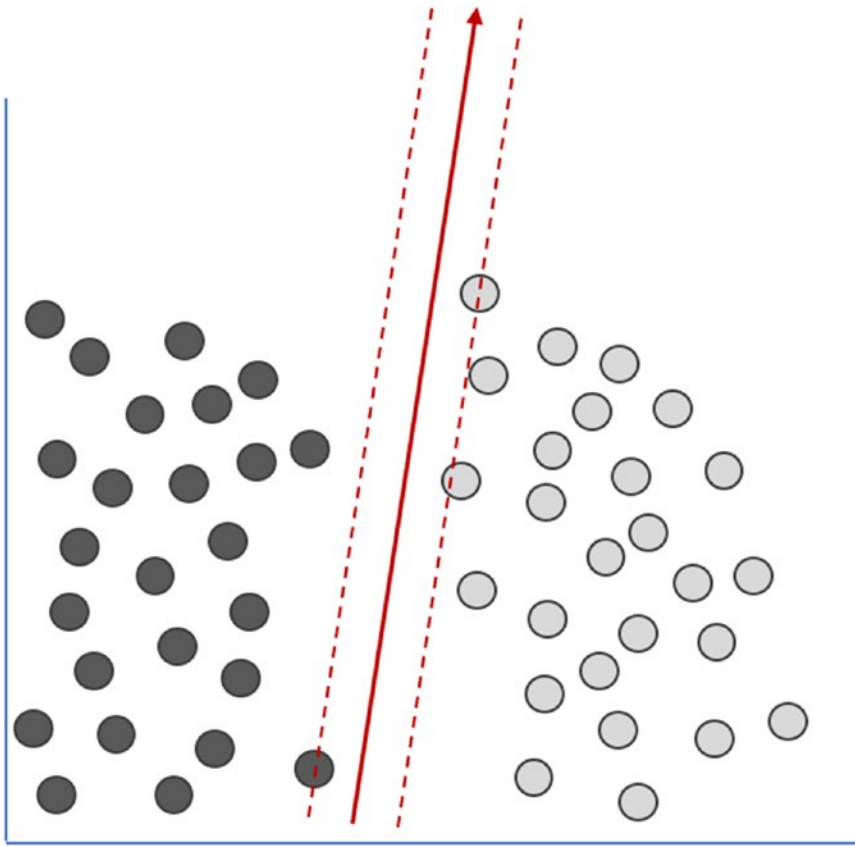


Figure 2-32. *The decision boundary drawn with support vectors*

Basically, a **support vector** is a vector parallel to the hyperplane that acts as the decision boundary, containing a point that is closest to the **hyperplane**, and helps establish a margin for the decision boundary. In this example, the hyperplane is a line because there are only two dimensions. In three dimensions, the hyperplane would be a plane, and in four dimensions, it would be a three-dimensional space, and so on.

The most optimal hyperplane would involve the support vectors establishing a maximum margin for the hyperplane. The example in Figure 2-32 is not optimal, so let's look for a more optimal hyperplane in Figure 2-33.

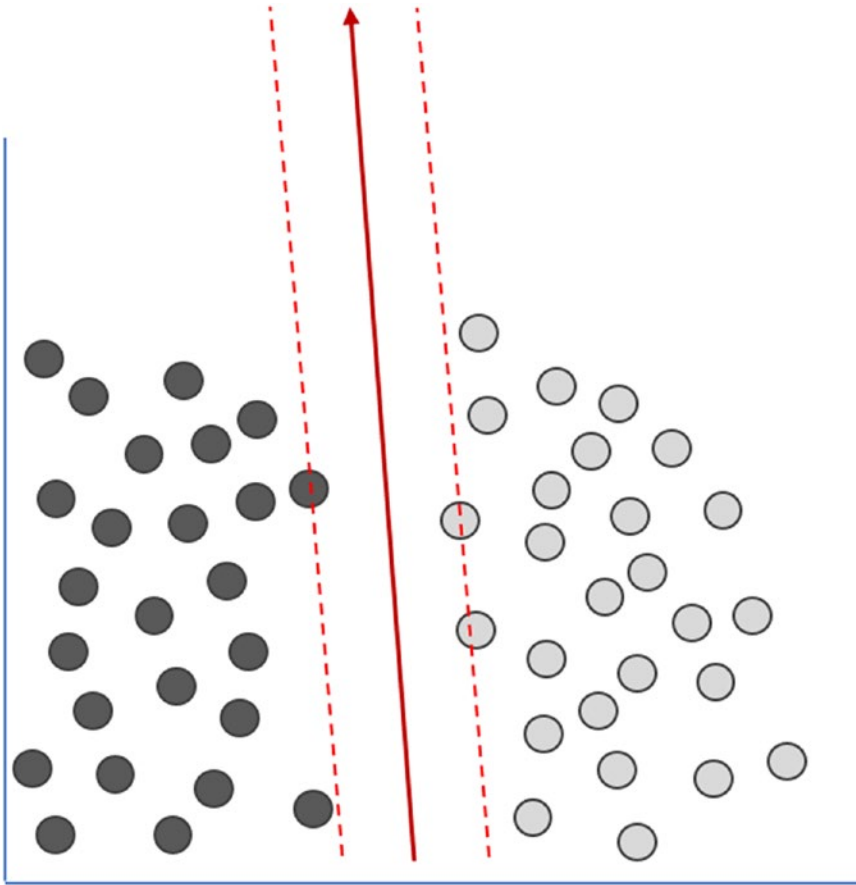


Figure 2-33. *A hyperplane with support vectors that allow for a larger margin*

With how the hyperplane is drawn, the points which their respective support vectors pass through are the closest to the hyperplane. This is a more optimal solution for a hyperplane since the margin for the hyperplane is much larger than in the previous example (Figure 2-32).

However, realistically, you will see hyperplanes that are more like Figure 2-34.

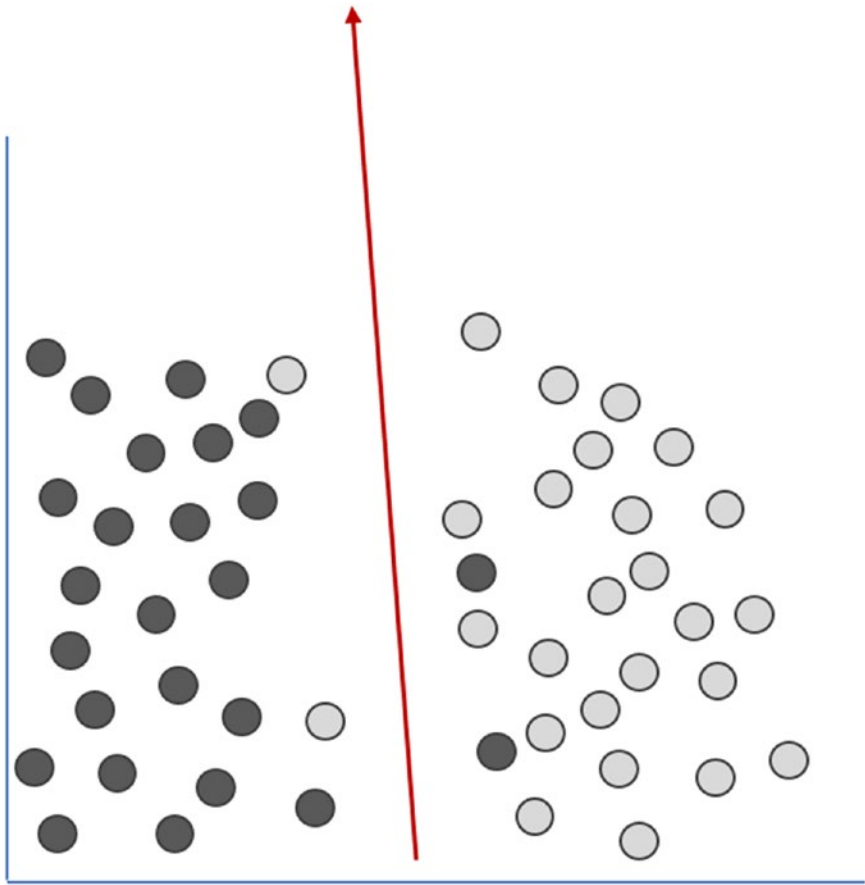


Figure 2-34. *A more realistic example of how a hyperplane functions*

There will always be outliers that prevent a clear distinction between two classifications. If you think back to the invasive fish example, there were some native fish that looked like invasive fish, and some invasive fish that looked like native fish.

Alternatively, Figure [2-35](#) shows a possible solution.

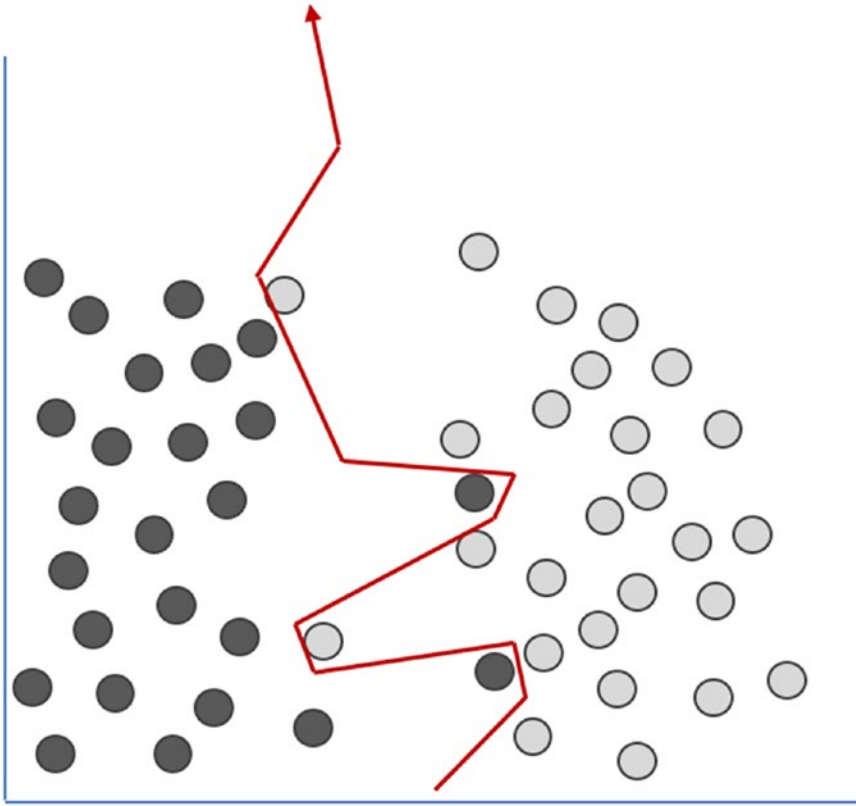


Figure 2-35. *An example of a hyperplane completely separating the two regions. However, this is an example of overfitting*

While this does count as a solution to the classification problem, this would lead to **overfitting**, resulting in another issue. If the SVM performs too well on the training data, it could perform worse on new data that contains different variations.

The decision boundaries won't be that simple either. You could run into situations such as the one shown in Figure 2-36.

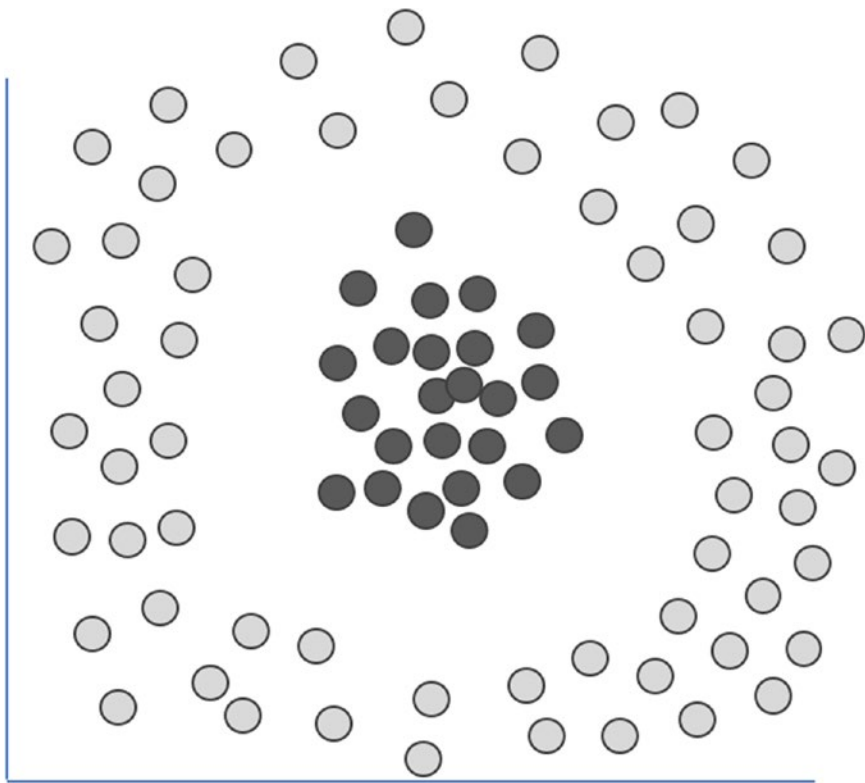


Figure 2-36. *A graph showcasing a different type of grouping of the data points*

You can't draw a line for this, so you have to think differently instead of using a linear SVM. Let's try to map the distances of each point from the center of the dark dots onto the 3D plane through some function (see Figure 2-37).

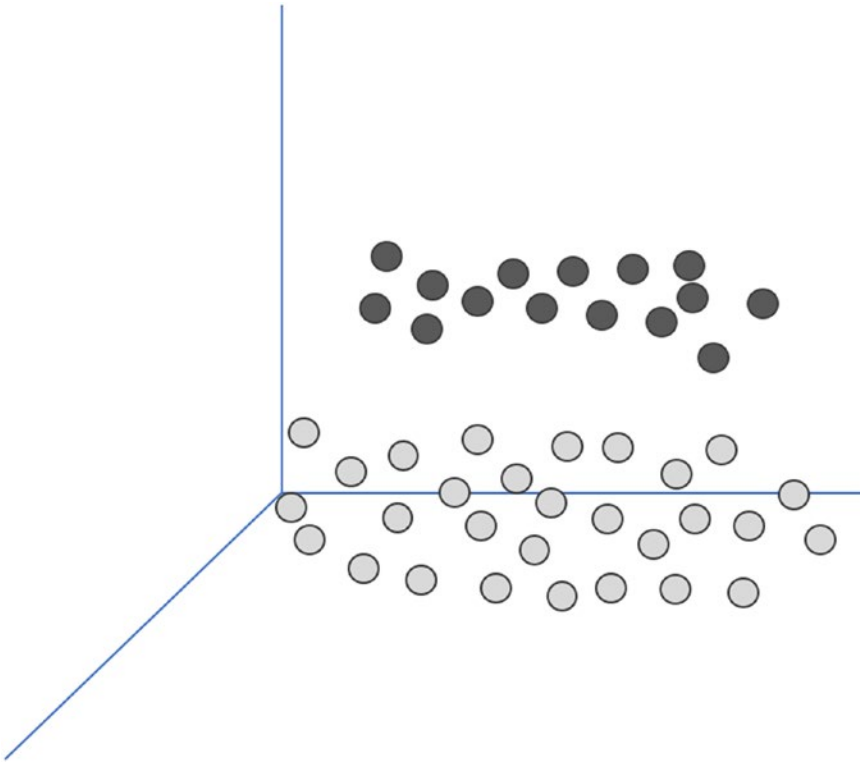


Figure 2-37. *Plotting the points onto the 3D plane shows that you can now separate the regions*

Now there is a clear separation between the two classes, and you can go ahead with separating the data points into two regions, as in [Figure 2-38](#).

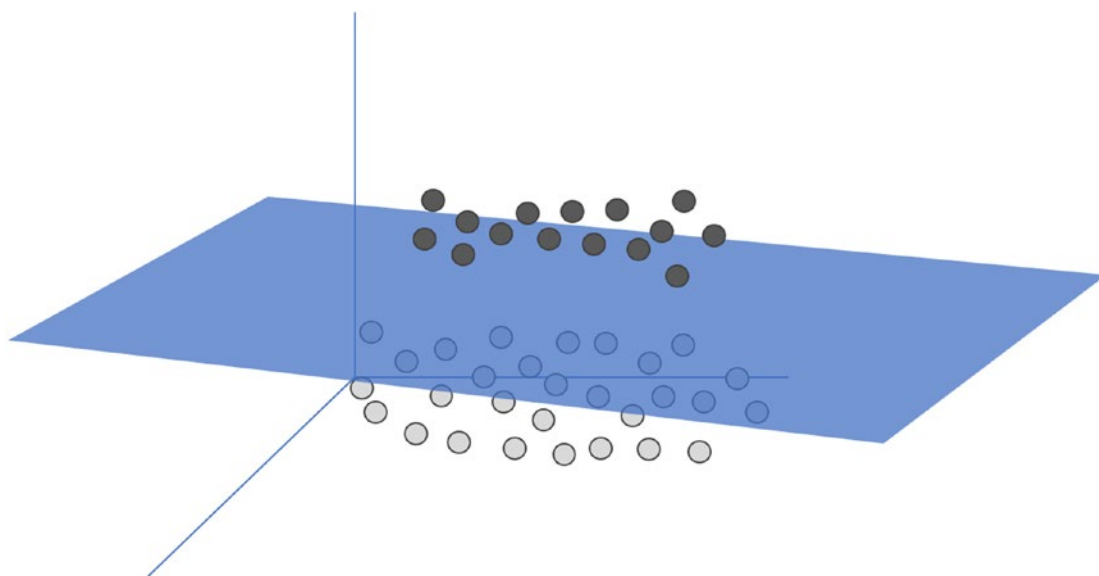


Figure 2-38. *The hyperplane now is an actual plane because of the added third dimension*

When you go back to the 2D representation of the points, you can see something like Figure [2-39](#).

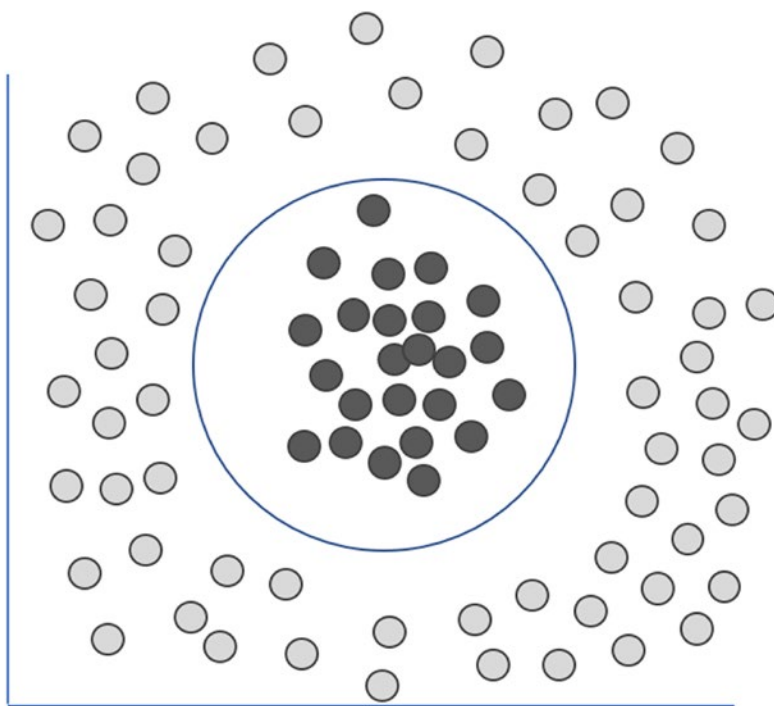


Figure 2-39. *This is what the hyperplane looks like when you go back to 2D*

What you just did was use a **kernel** to transform the data into another dimension where there is a clear distinction between the classes of data. This mapping of data is called a **kernel trick**. There are different types of kernels, including the **linear kernel** you saw in the earlier examples. Other types of kernels include **polynomial kernels**, which map the data to some n th dimension using a polynomial function, and **exponential kernels**, which map the data according to an exponential function.

Another term to cover is **regularization**, a parameter that tells the SVM how much you want to avoid misclassifications. **Lower regularization** values lead to graphs like the one you saw earlier where there were a few outliers on either side of the hyperplane. **Higher regularization** values lead to graphs where you saw the hyperplane separate every single point, at the cost of possibly overfitting on the data.

Gamma tells the SVM how much to consider points farther away from the region of separation between the classes. **Higher gamma** values tell the SVM to only consider nearby points, while **lower gamma** values tell the SVM to also consider the points farther away.

Finally, the **margin** is the separation between each class and the hyperplane. As discussed earlier, an **ideal margin** involves the maximum equidistant separation of each of the closest from the hyperplane. A **bad margin** or **suboptimal margin** has the hyperplane too close to one class or the distance not be as far as it can be to the hyperplane for each point or support vector.

As for the one-class support vector machine, Figure 2-40 shows what the graph would look like.

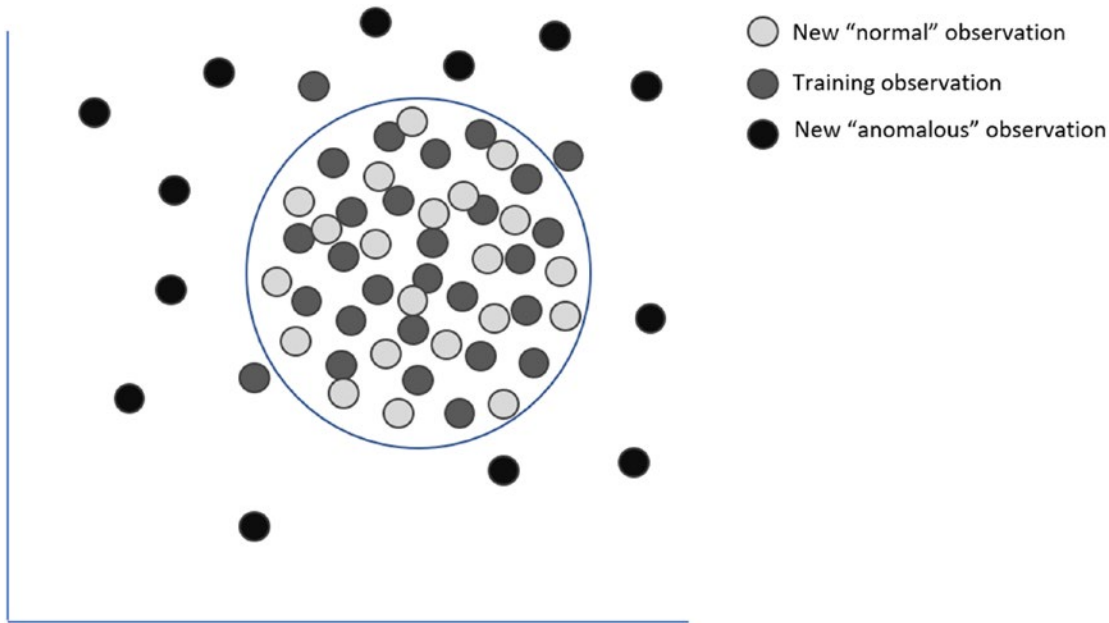


Figure 2-40. *An example of the decision boundary for a one-class support vector machine*

During training, the OC-SVM learns the decision boundary for normal observations, accounting for a few outliers. If **novelties**, new data points that the model has never seen before, fall within this decision boundary, they are considered normal by the model. If they fall outside of the boundary, they are considered anomalous. This technique is an example of semi-supervised novelty detection, where the goal is to train the model on normal data, and then it attempts to find anomalies in new data.

By doing so, the OC-SVM can capture the shape of the data pretty well thanks to the decision boundary that captures most of the training observations.

Anomaly Detection with OC-SVM

Now that you know more about how SVMs work, let's get started by applying a one-class SVM to the KDDCUP 1999 data set.

Import your modules and load up the data set (see Figure 2-41 and Figure 2-42).

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.svm import OneClassSVM

%matplotlib inline
```

Figure 2-41. *Importing your modules for the OC-SVM*

```
columns = ["duration", "protocol_type", "service", "flag", "src_bytes",
"dst_bytes", "land", "wrong_fragment", "urgent",
        "hot", "num_failed_logins", "logged_in", "num_compromised",
"root_shell", "su_attempted", "num_root",
        "num_file_creations", "num_shells", "num_access_files",
"num_outbound_cmds", "is_host_login",
        "is_guest_login", "count", "srv_count", "error_rate",
"srv_error_rate", "rerror_rate", "srv_rerror_rate",
        "same_srv_rate", "diff_srv_rate", "srv_diff_host_rate",
"dst_host_count", "dst_host_srv_count",
        "dst_host_same_srv_rate", "dst_host_diff_srv_rate",
"dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate",
        "dst_host_error_rate", "dst_host_srv_error_rate",
"dst_host_rerror_rate", "dst_host_srv_rerror_rate", "label"]

df = pd.read_csv("datasets/kdd_cup_1999/kddcup.data/kddcup.data.corrected",
sep=",", names=columns, index_col=None)
```

Figure 2-42. *Defining the columns for the data set, and importing the data set into the data frame variable df*

Now, let's move on to filtering out all the normal data entries. You will make two data frames that consist of normal entries and an equal mix of anomalies and normal data entries.

Run the code in Figure 2-43.

```
df = df[df["service"] == "http"]
df = df.drop("service", axis=1)
columns.remove("service")

novelties = df[df["label"] != "normal."]
novelties_normal = df[150000:154045]

novelties = pd.concat([novelties, novelties_normal])
normal = df[df["label"] == "normal."]
```

Figure 2-43. *Filtering out the anomalies and the normal data points to construct a new data set that is a mixture of the two*

Figure 2-44 shows the shapes of the two data frames.

```
In [308]: 1 print(novelties.shape)
          2 print(normal.shape)

(8090, 41)
(619046, 41)
```

Figure 2-44. *Printing out the shapes of the novelty and normal data sets*

The first half of the data frame “novelties” consists of anomalies, while the latter half consists of normal data entries.

Now you move on to encoding all the categorical values in the data frames (see Figure 2-45).

```

for col in normal.columns:
    if normal[col].dtype == "object":
        encoded = LabelEncoder()
        encoded.fit(normal[col])
        normal[col] = encoded.transform(normal[col])

for col in novelties.columns:
    if novelties[col].dtype == "object":
        encoded2 = LabelEncoder()
        encoded2.fit(novelties[col])
        novelties[col] = encoded2.transform(novelties[col])

```

Figure 2-45. Applying the label encoder to the data sets

Now run the code in Figure 2-46 to set up your training, testing, and validation sets.

```

for f in range(0, 10):
    normal = normal.iloc[np.random.permutation(len(normal))]

df2 = pd.concat([normal[:100000], normal[200000:250000]])
df_validate = normal[100000:150000]
x_train, x_test = train_test_split(df2, test_size = 0.2,
    random_state = 42)
x_val = df_validate

```

Figure 2-46. Shuffling the entries in the normal data set, and defining the training, testing, and validation sets

Figure 2-47 shows the shapes of the data sets.

```
In [12]: 1 print("Shapes:\nx_train:{}\n".format(x_train.shape))
          2 print("x_test:{}\n".format(x_test.shape))
          3 print("x_val:{}\n".format(x_val.shape))
          4

Shapes:
x_train:(120000, 41)

x_test:(30000, 41)

x_val:(50000, 41)
```

Figure 2-47. *Printing the output shapes of the training, testing, and validation sets*

You are only using a subset of the entire data set to train the model on because the larger the training data, the longer it takes for the OC-SVM to train.

Run the code in Figure 2-48 to declare and initialize the model.

```
ocsvm = OneClassSVM(kernel='rbf', gamma=0.00005, random_state =
42, nu=0.1)
```

Figure 2-48. *Defining your OC-SVM model*

By default, the **kernel** is set to 'rbf', meaning radial basis function. It is similar to the circular decision boundary that you saw in the earlier examples, and you use it here because you want to define a circular boundary around a set of regions that contain normal data. As seen in the earlier examples, any points that fall outside of the region are to be considered anomalies. **Gamma** tells the model how much you want to consider points further from the hyperplane. Since it is pretty small, this means you want to emphasize the points farther away. The **random_state** is just a seed for initializing the random number generator, similar to the isolation forest model. The next parameter, **nu**, specifies how much of the training set contains outliers. Again, you set this to 0.1, similar to the isolation forest model. This acts similar to the regularization parameter that you saw earlier, since it tells the model approximately how many data points you expect the model to misclassify.

Now let's train the model and evaluate predictions (see Figure 2-49).

```
ocsvm.fit(x_train)
```

Figure 2-49. *Training the OC-SVM model on the training data*

One thing to note is that you can't get the values for an AUC curve for `x_test` and `x_validation` since they comprise entirely of normal data values. You can't get values for true negative or for false positive since there are no anomalies in the data set to classify falsely as normal or correctly as anomalies.

However, you can still measure the accuracy of the model on the test and validation sets. Even though accuracy is not the best metric to go by, it can still give you a good indicator of the model's performance.

Also one thing to note: Accuracy in this case is a measure of the percentage of data points in the predictions that are normal data points. Remember, you assumed that around 10% of the data points in the data set are anomalies, so the most optimal "accuracy" to obtain is 90%.

Run the code in Figure 2-50.

```
preds = ocsvm.predict(x_test)
score = 0
for f in range(0, x_test.shape[0]):
    if(preds[f] == 1):
        score = score + 1

accuracy = score / x_test.shape[0]
print("Accuracy: {:.2%}".format(accuracy))
```

Figure 2-50. *Making predictions and generating the "accuracy" score*

```
In [33]: 1
          2 preds = ocsvm.predict(x_test)
          3 score = 0
          4 for f in range(0, x_test.shape[0]):
          5     if(preds[f] == 1):
          6         score = score + 1
          7
          8 accuracy = score / x_test.shape[0]
          9 print("Accuracy: {:.2%}".format(accuracy))

Accuracy: 89.09%
```

Figure 2-51. The resulting output accuracy for the testing data set

Figure 2-51 shows that the accuracy is about 89.1%, which is pretty good considering that you assumed 10% of the data would misclassify.

Let's run the code on `x_validation` this time (see Figure 2-52).

```
preds = ocsvm.predict(x_val)
score = 0
for f in range(0, x_val.shape[0]):
    if(preds[f] == 1):
        score = score + 1

accuracy = score / x_val.shape[0]
print("Accuracy: {:.2%}".format(accuracy))
```

Figure 2-52. Generating the accuracy score for the validation set

```

In [34]: 1 preds = ocsvm.predict(x_val)
          2 score = 0
          3 for f in range(0, x_val.shape[0]):
          4     if(preds[f] == 1):
          5         score = score + 1
          6
          7 accuracy = score / x_val.shape[0]
          8 print("Accuracy: {:.2%}".format(accuracy))

Accuracy: 89.49%

```

Figure 2-53. The resulting percentage of data points in the predictions that were considered normal

This time the accuracy was even better at around 89.5% (Figure 2-53).

Now to test on the novelties data set. This time, you can find the AUC score because there is a 50-50 split between anomalies and normal data. The other two data sets, `x_test` and `x_validation`, only had normal data, but this time it is possible for the model to classify false positives and true negatives.

Run the code in Figure 2-54.

```

from sklearn.metrics import roc_auc_score

preds = ocsvm.predict(novelties)
matches = novelties["label"] == 4

auc = roc_auc_score(preds, matches)
print("AUC: {:.2%}".format (auc))

```

Figure 2-54. The code to generate the AUC score

```
In [47]: 1 from sklearn.metrics import roc_auc_score
          2
          3 preds = ocsvm.predict(noverties)
          4 matches = noverties["label"] == 4
          5
          6 auc = roc_auc_score(preds, matches)
          7 print("AUC: {:.2%}".format (auc))

AUC: 95.83%
```

Figure 2-55. The generated AUC score from the predictions on the novelty set

Figure 2-55 shows the score. That's pretty good for an AUC score!

Let's look at the distribution of predictions in Figure 2-56.

```
plt.figure(figsize=(10,5))
plt.hist(preds, bins=[-1.5, -0.5] + [0.5, 1.5], align='mid')
plt.xticks([-1, 1])
plt.show()
```

Figure 2-56. Code to display a graph that shows the distributions for the predictions

```
In [48]: 1 plt.figure(figsize=(10,5))
2         plt.hist(preds, bins=[-1.5, -0.5] + [0.5, 1.5], align='mid')
3         plt.xticks([-1, 1])
4         plt.show()
```

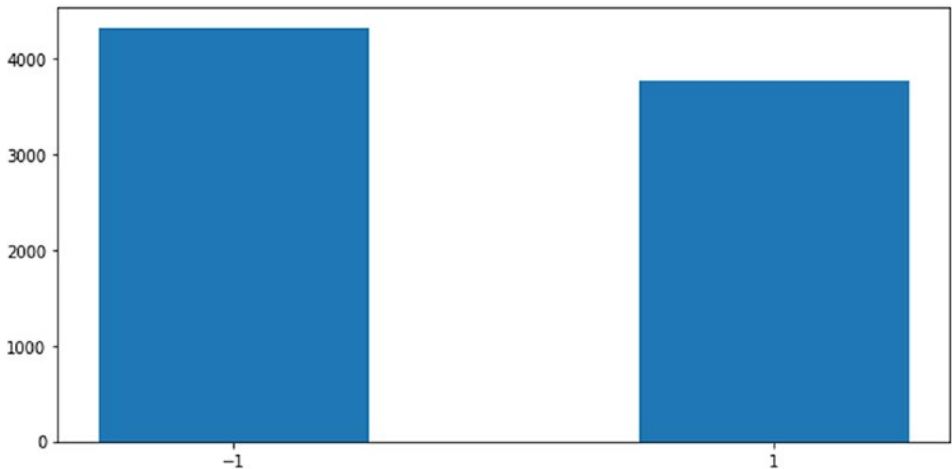


Figure 2-57. The resulting output. 1 stands for normal data points, and -1 stands for anomaly data points

As you can see in Figure 2-57, the model ended up predicting more anomalies than normal data points, but from what the AUC tells us, it managed to classify most of the data entries correctly.

Hopefully by now you will have gained a better understanding of what an OC-SVM is and how to apply it. Remember, OC-SVM works well for multi-dimensional data (in this case, you had 41 columns after dropping the service column) and can be used for **semi-supervised anomaly detection** when applied in the manner implemented in this section.

Summary

In this chapter, we discussed traditional methods of anomaly detection and how they can be used to implement anomaly detection in an **unsupervised** and **semi-supervised** manner.

In the next chapter, we will look at the advent of deep learning networks.

CHAPTER 3

Introduction to Deep Learning

In this chapter, you will learn about deep learning networks. You will also learn how deep neural networks work and how you can implement a deep learning neural networks using Keras and PyTorch.

In a nutshell, the following topics will be covered throughout this chapter:

- What is deep learning?
- Intro to Keras: A simple classifier model
- Intro to PyTorch: A simple classifier model

What Is Deep Learning?

Deep learning is a special subfield of machine learning that deals with different types of artificial neural networks. Drawing inspiration from the structure and functionality of a brain, artificial neural networks at their core are layers of interlinked, individual units call neurons that each perform a specific function given input data.

In “deep” learning specifically, some of the best models consist of dozens of layers and millions of neurons, and have been trained on multiple gigabytes of data. Generally, deep learning models don’t always need to be this big to perform well on certain tasks, and the tasks that the large models are expected to perform are complex, ranging from outlining a wide variety of objects within an image to generating summaries of articles.

Thanks to recent increases in the computational power and availability of GPUs (graphics processing units), anyone with access to a decent enough GPU can train their own deep learning models, keeping in mind that larger models might require more GPU resources such as memory.

Today, deep learning is taking the world by storm thanks to the extreme versatility and performance that it offers. More traditional models in machine learning have a problem where adding more training samples leads to a plateau in performance, but that problem doesn't exist with deep learning. Instead, deep learning models get better and better with more samples, meaning they scale far better in terms of data set size and gain better performance as a result. Deep learning models can be applied to nearly any task with resounding success, and so are employed in the fields of cybersecurity, meteorology, finances and stock markets, speech recognition, medicine, search engines, etc. What exactly about deep learning makes it so great? First, let's take a look at what an artificial neural network is.

Artificial Neural Networks

Artificial neural networks are layers of interconnected nodes, or artificial neurons, that function in a way inspired by biological neural networks. Figure 3-1 shows an example of a neuron.

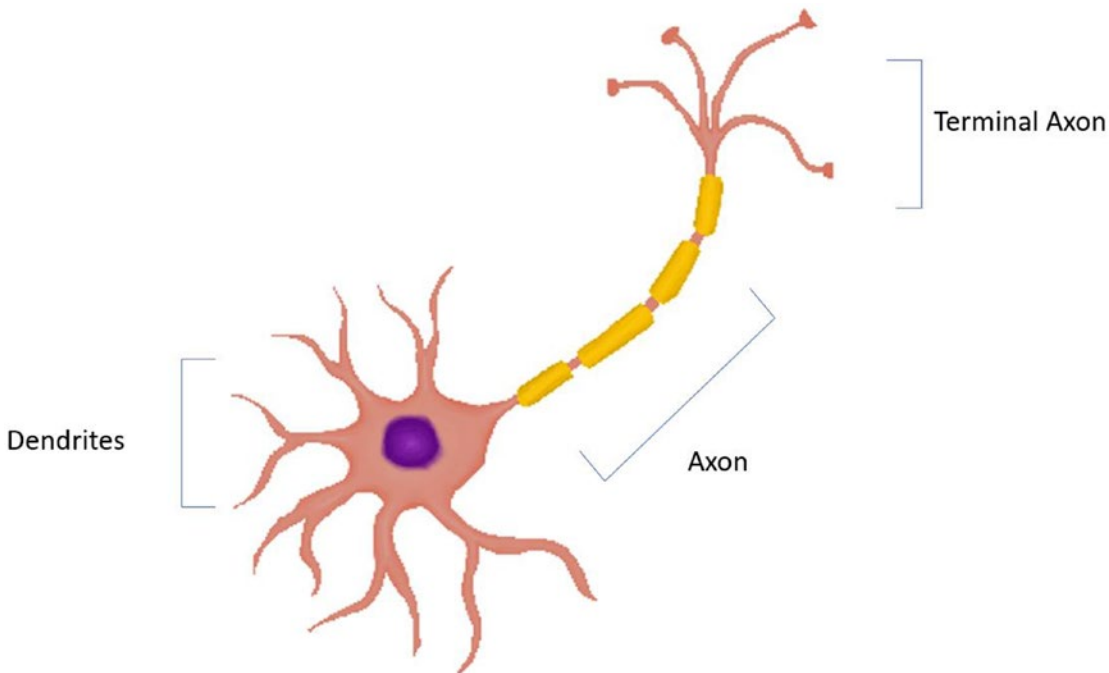


Figure 3-1. *An example of what a neuron can look like*

Inputs are taken in through the dendrites, after which the neuron decides whether or not to fire. Upon firing, the neuron sends a signal down the axon to its terminal axons, where the signals are output to any other neurons. This transfer of signals is called a synapse, which is modeled in Figure 3-2.

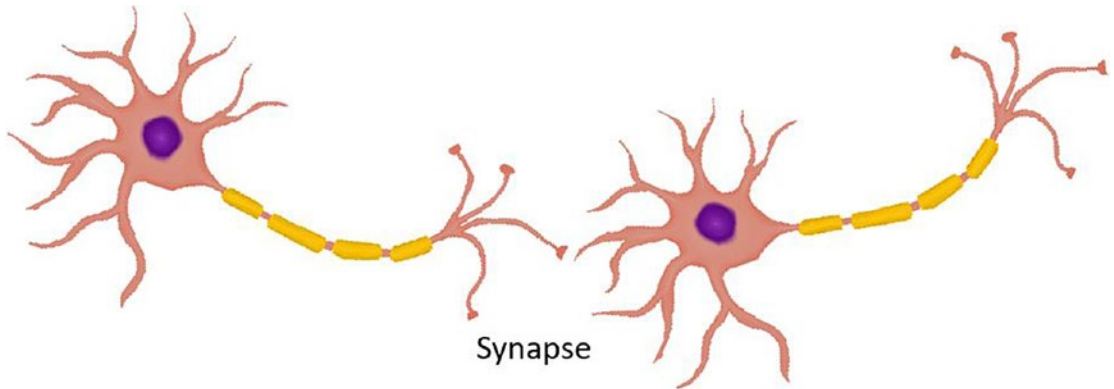


Figure 3-2. How two neurons might connect to form a chain and transfer signals through that connection. The terminal axon of the first neuron connects to the dendrites of the second neuron

We use a similar concept in artificial neural networks (Figure 3-3).

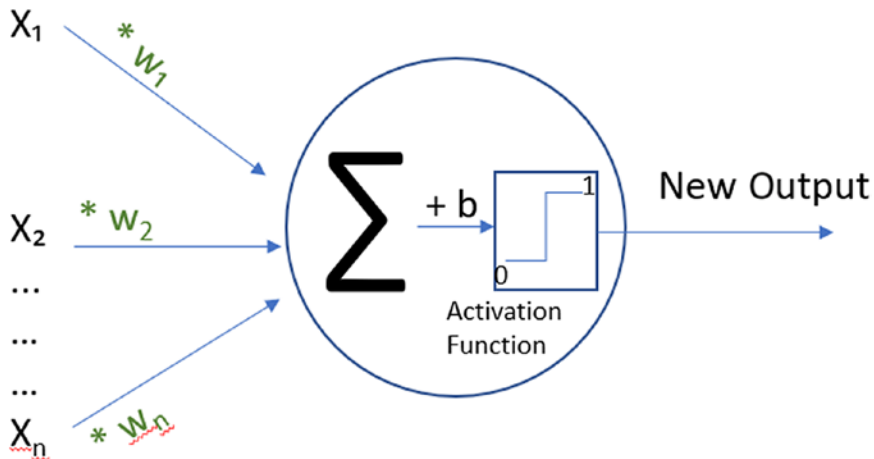


Figure 3-3. How an artificial neuron in an artificial neural network can function. This mimicry of the biological neuron is the basis of artificial neural networks

In the case of this artificial neuron, we find the dot product between the input vector **X** and the weight vector **W**. **X** represents the input data, and **W** represents the list of weights that this node carries to multiply with the input vector. Recall that the dot product is when each element in the vector is multiplied with the corresponding element in the second vector, as in Figure 3-4.

$$\langle a, b, d \rangle \cdot \langle e, f, g \rangle = ae + bf + dg$$

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \cdot \begin{bmatrix} e \\ f \\ g \end{bmatrix} = ae + bf + cg$$

Figure 3-4. This is how dot product works. Shown here is an example with two different types of vector notation

Both are different ways to represent a vector, although the second method is ideal considering your data and weights would most likely take the shape of a matrix.

After that, there is an optional bias function where the value **b** (called **bias**) is added to the dot product result. From there, it passes through an **activation function** that decides if the entire node sends data or not. In this case, the activation function only varies between 0 and 1 depending on whether or not the dot product plus the bias reaches a certain value or not (threshold). It is possible to have other activation functions such as a sigmoid function, which outputs some value between 0 and 1.

Calling the output *y* and the input *x*, the basic function for each node can be represented by the equation in Figure 3-5.

$$y = f\left(\sum_{i=1}^n w_i x_i + b\right)$$

Figure 3-5. An equation that captures the basic functionality of an artificial neuron. In this case, *f(x)* is an activation function

An artificial neural network is comprised of interlinked layers of these nodes and can look like Figure 3-6.

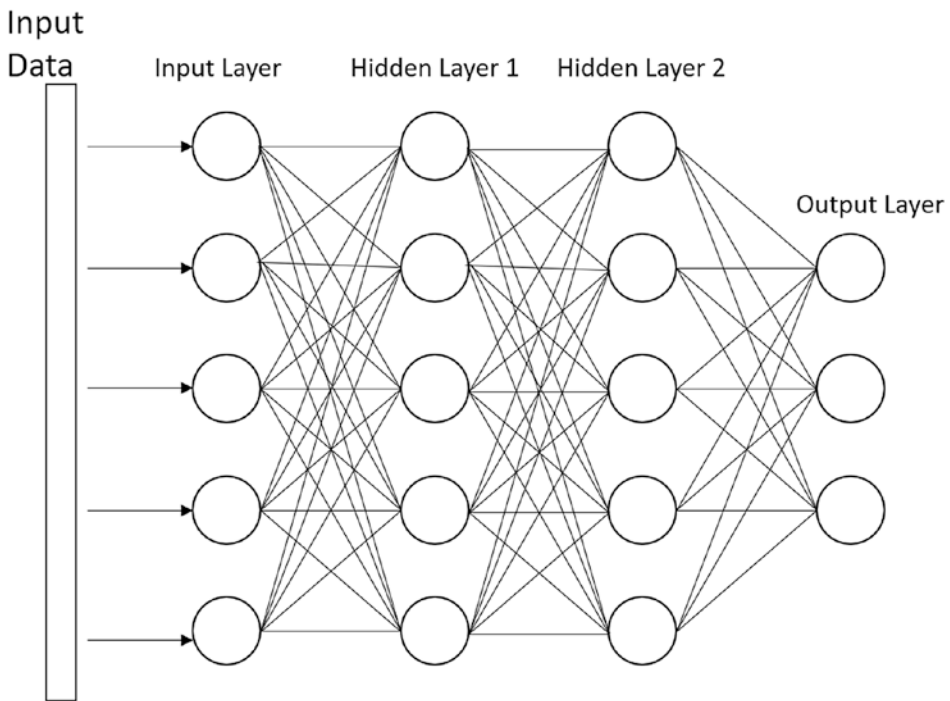


Figure 3-6. *An example of what an artificial neural network can look like*

A **hidden layer** is one that is between the input layer and the output layer. There can be multiple hidden layers in a network. Now that you’ve seen what an artificial neural network can look like, let’s take a look at how the data can flow through this network. First, we start with nothing but the input data in the network, and assume that neurons only wholly activate (neurons can partially activate depending on the activation function, but in this example each neuron either outputs a 1 or a 0) (Figure 3-7).

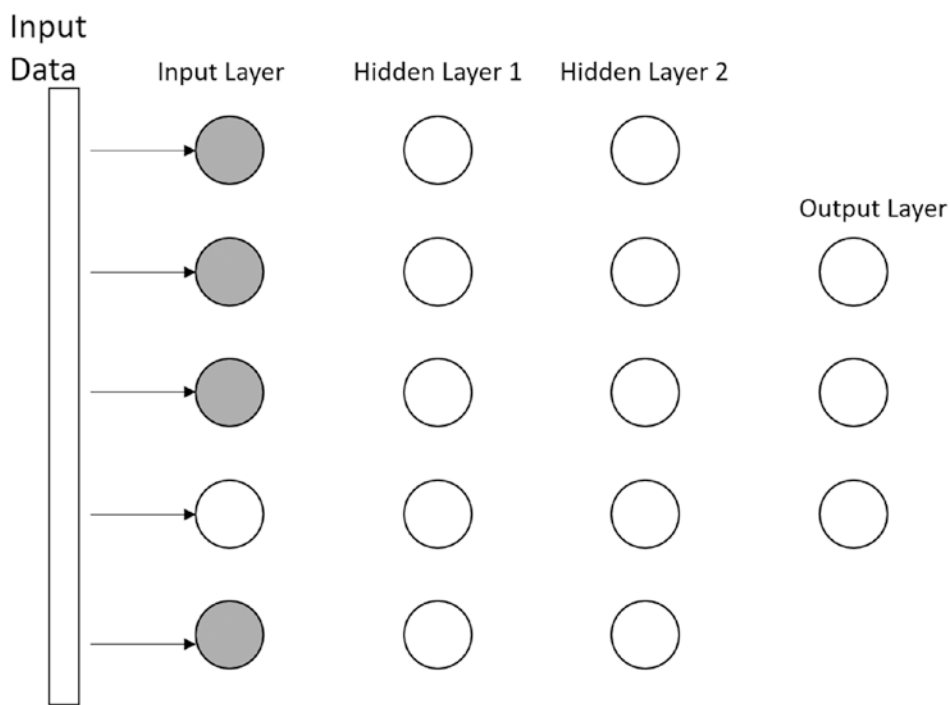


Figure 3-7. The input data runs through the input layer, and selective nodes fire based on the input received

The input layer takes all of the corresponding inputs and produces an output that is linked to the first hidden layer. The outputs of the nodes that activate in the input layer are now the inputs of the hidden layer, and the new data flows correspondingly (Figure 3-8).

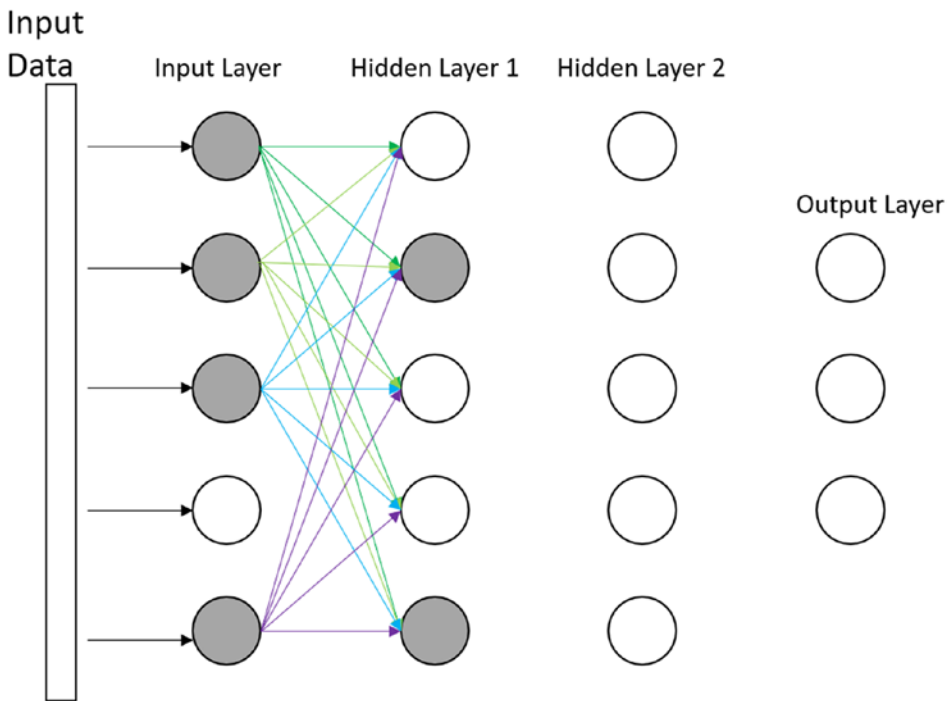


Figure 3-8. The outputs from the activated neurons in the input layer pass on to the first hidden layer. These outputs are now the inputs of the next layer, and selective neurons fire based on this input

Hidden layer 1 processes the data in a similar fashion to the input layer, just with different parameters for activation function, weight, bias, etc. The data passes through this layer and the output of this layer becomes the input for the next hidden layer. In this case, only two nodes activate based on the input from the previous layer (Figure 3-9).

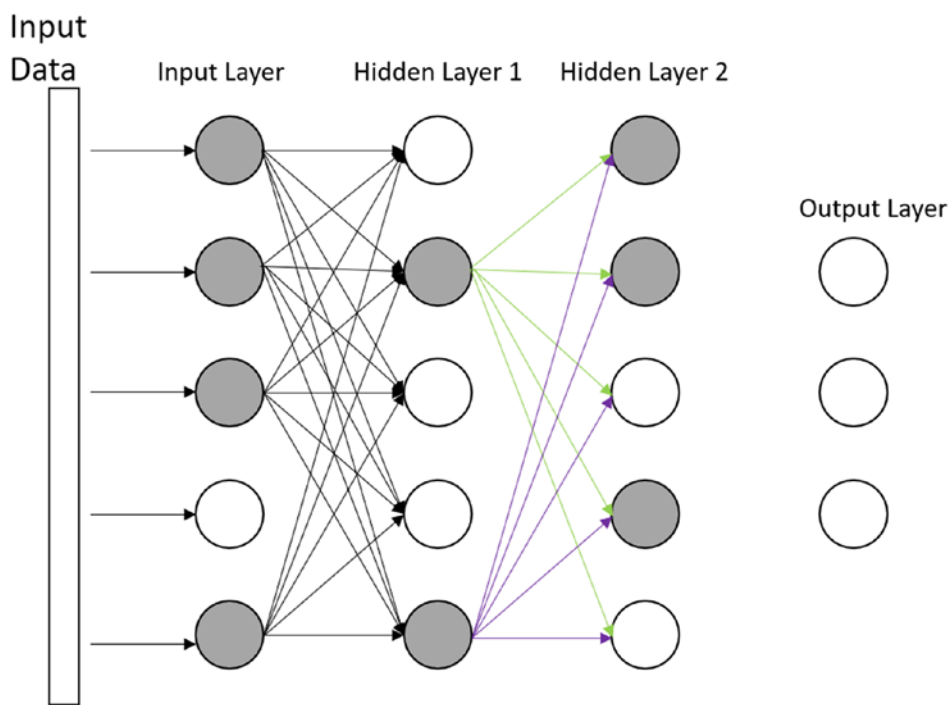


Figure 3-9. *This process repeats with hidden layer 2*

Hidden layer 2 processes the data and sends the data to a new layer called the output layer, where only one of the nodes in the layer will be activated. In this case, the first node in the output layer is activated (Figure 3-10).

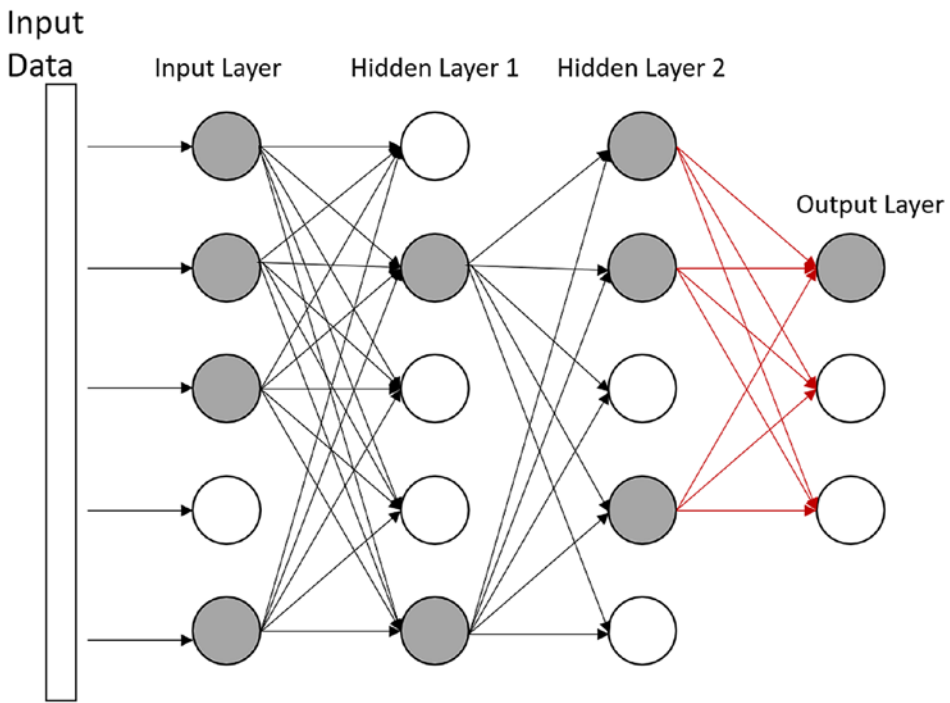


Figure 3-10. Finally, the data from the second hidden layer goes to the output layer, where one neuron fires in this case

The nodes in the output layer can represent the different labels that you want to give to the input data. For example, in the iris dataset, you can take various measurements of an iris flower and train an artificial neural network on this data to classify the species of the flower.

Upon initialization, the weights of the model will be far from ideal. Throughout the training process, the data flow from the model goes forward (left to right from input to output), and then backwards in what is known as **backpropagation** to recalculate the weights and biases for each activated node.

In backpropagation, a **cost function** takes into account the model's predictions for one pass of the training data through the network and what the actual predictions should be. The cost function gives you an indicator of how good the model's weights are at predicting the correct outcome. For this example, assume that Figure 3-11 shows the formula of the cost function.

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n (h_{\theta}(x^i) - y^i)^2$$

Figure 3-11. The formula for the mean squared error cost function

This cost function is called the **mean squared error**, named so because the function given input θ , the weights, finds the average difference squared between the predicted value and the actual value. The parameter h_{θ} represents the model with the weight parameter θ passed in, so $h_{\theta}(x^i)$ gives the predicted value for x^i with model's weights θ . The parameter y^i represents the actual prediction for the data point at index i . If the parameter you are passing in includes both weight and bias, then it will look more like Figure 3-12.

$$J(w, b) = \frac{1}{n} \sum_{i=1}^n (h_{w,b}(x^i) - y^i)^2$$

Figure 3-12. A formula for the mean squared error cost function, with more specific notation separating the weights and the biases

Note that $h_{w,b}(x^i)$ will have the formula in Figure 3-13.

$$h(w, b) = wx^i + b$$

Figure 3-13. An elaboration on what the function $h(w, b)$ means in Figure 3-12

The cost function reflects the overall performance of the model with the current weight parameter, so the most ideal value output from the cost function will be as small as possible. Since the cost function is a measure of how far the model's predictions are from the actual value, you want to make the output from the cost function as small as possible since that means your predictions were almost what the actual prediction should be.

To minimize the cost function, you need to tell the model how to adjust the weights, but how do you do that? If you think back to calculus, optimization problems involved finding the derivative and solving for the critical points (points where the derivative of the original equation is 0). In your case, you want to find the **gradient**, which can be

thought of as similar to the derivative but in a multi-dimensional setting, and adjust the weights in a direction that would change the gradient so it approaches 0.

There are several optimization algorithms to help the model achieve the optimal weights including **gradient descent**. Gradient descent is an optimization algorithm that finds the gradient of the cost function and takes a single step in the direction of the local minimum to generate values to use to adjust the weights and biases.

How much of a step you take is controlled by the **learning rate**. The bigger the learning rate, the larger the step you take at each iteration, and the quicker the local minimum is approached. The smaller the learning rate, the longer the training takes since the steps are smaller. However, a problem with too large of a learning rate is that it could overshoot the local minimum entirely, leading to a complete failure to ever reach the local minimum. Too small of a learning rate and the local minimum might take way too long to reach. When the model starts to reach an ideal level of performance, the gradients should be approaching 0 since the weights would have the cost function reach a local minimum, signifying that the differences between the model's predictions and the actual predictions are very small.

In a process called **backpropagation**, the gradients are calculated and the weights are adjusted for each node in a layer, before the same process is done for the layer before that until all of the layers have had their weights adjusted. The entire process of passing the data through the model and backpropagating to readjust the weights is what comprises the training process of a model in deep learning.

While the entire training process may sound complicated and computationally heavy, GPUs help train the models much quicker because they are optimized to perform the matrix calculations required by graphics processing.

Now that you know more about what deep learning is and how artificial neural networks operate, a question might arise on **why** we should use deep learning for anomaly detection.

First of all, thanks to the advancements in GPU technology, we can train deep learning models that are far deeper (many layers with lots of parameters) and on huge data sets. This in itself leads to incredible performances by the networks and allows the model to have much more powerful applications.

Not only has this led to a diverse set of models that are each suited for different applications (image classification, video captioning, object detection, language translation, generative models that can summarize articles, etc.), but the models keep getting better and better at their respective tasks.

The models are also far more scalable than their traditional counterparts, since deep learning models don't hit a plateau in training accuracy as the number of data entries increases, meaning we can apply deep learning models to massive volumes of data. This attribute of deep learning models pairs very well with the trend of big data in today's society.

In this chapter, you will look at applying deep learning models to classifying handwritten digits as an introduction to using two great, popular deep learning frameworks in Python: **Keras**, with a TensorFlow backend, and **PyTorch**. These frameworks help you create customized deep learning models in just a few dozen lines of code as opposed to creating them entirely from scratch.

Keras is a high-level framework that lets you quickly create, train, and test powerful deep learning models while abstracting all of the little details away for you. **PyTorch** is more of a low-level framework, but it doesn't carry with it the amount of syntax that **TensorFlow** (a much more popular deep learning framework) does. Compared to Keras, however, there are still more things that you must define since it's no longer abstracted away for you.

Using PyTorch over TensorFlow or vice-versa is more of a personal preference, but PyTorch is easier to pick up. Both offer very similar functionality, and if there are any functions that TensorFlow has that PyTorch doesn't, you can still implement them using the PyTorch API.

Another note to make is that TensorFlow has integrated Keras into its API, so if you want to use TensorFlow in the future, you can still build your models using `tf.keras`.

Intro to Keras: A Simple Classifier Model

Before you get started, it is recommended that you have the GPU version of TensorFlow installed along with all of its dependencies, including CUDA and cuDNN. While they are not necessarily required to train deep learning models, having a GPU helps to massively reduce training time. Both TensorFlow and PyTorch utilize CUDA and cuDNN to access the GPU while training, and Keras runs on top of TensorFlow.

If you have any questions about Keras, feel free to refer to Appendix A to get a better understanding of how Keras works and of the functionality that it offers.

Here are the exact versions of the necessary Python 3 packages used:

- tensorflow-gpu version 1.10.0
- keras version 2.0.8
- torch version 0.4.1 (this is PyTorch)
- CUDA version 9.0.176
- cuDNN version 7.3.0.29

You will create, train, and evaluate a deep learning architecture known as a convolutional neural network (CNN) in Keras using the MNIST data set. You don't need to download this data set since it is included within TensorFlow.

The MNIST data set, or the Modified National Institute of Standards and Technology data set, is a large collection of handwritten images used to train computer vision and image processing models such as the CNN. It is a common data set to start with and is basically like the “hello world” data set of computer vision.

The data set contains 60,000 training images and 10,000 testing images of handwritten digits 0-9, each with a dimension of 28x28 pixels.

First, import all the dependencies (Figure 3-14).

```
import tensorflow as tf
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Input
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
import numpy as np
```

Figure 3-14. *Importing the modules needed to create the model*

Now define some variables that you will use later (Figure 3-15).

```
batch_size = 128  
n_classes = 10  
n_epochs = 15  
  
im_row, im_col = 28, 28
```

Figure 3-15. *Variables to use later*

One pass of the entire data set through the model is called an **epoch**. The **batch size** is how many data entries pass through the model in one iteration. In this case, the training data passes through the model 128 entries at a time until all of the entries have passed through, marking the end of one epoch. The number of classes is 10 to represent each of the 10 digits from 0-9. These variables are also known as **hyperparameters**, parameters that are set before the training process.

Let's create your training and testing data sets. One thing to note is that you can use data frames, arrays, matrices, etc. in Keras to serve as your data sets. Run the code in Figure 3-16.

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Figure 3-16. *Define the training and testing data sets*

You can use matplotlib to see what one of these images looks like. Run the code in Figure 3-17 and see the results in Figure 3-18.

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.imshow(x_train[1], cmap='gray')
plt.show()
```

Figure 3-17. *Importing `matplotlib.pyplot` to see what these training images look like*

```
In [68]: 1 import matplotlib.pyplot as plt
          2 %matplotlib inline
          3
          4 plt.imshow(x_train[1], cmap='gray')
          5 plt.show()
```

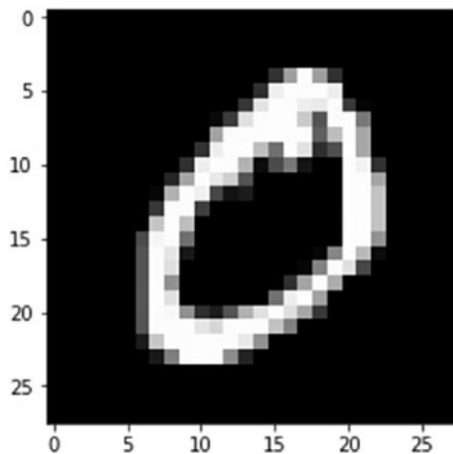


Figure 3-18. *The output of running the code in figure 3-17*

You can enter anywhere from 0 to 59,999 to visualize a sample in `x_train`.

Just looking at 10 examples of the digit 1, you can see there is plenty of variation in the data set (see Figure 3-19 and Figure 3-20).

```
fig = plt.figure(figsize=(15,10))

i = 0
for f in range(0, y_train.shape[0]):
    if(y_train[f] == 1 and i < 10):
        plt.subplot(2, 5, i+1)
        plt.imshow(x_train[f], cmap='gray')
        plt.xticks([])
        plt.yticks([])
        i = i + 1

plt.show()
```

Figure 3-19. Code to generate a plot that shows some example images for a specific class


```
In [103]: 1 fig = plt.figure(figsize=(15,10))
2
3 i = 0
4 for f in range(0, y_train.shape[0]):
5     if(y_train[f] == 1 and i < 10):
6         plt.subplot(2, 5, i+1)
7         plt.imshow(x_train[f], cmap='gray')
8         plt.xticks([])
9         plt.yticks([])
10        i = i + 1
11
12 plt.show()
```

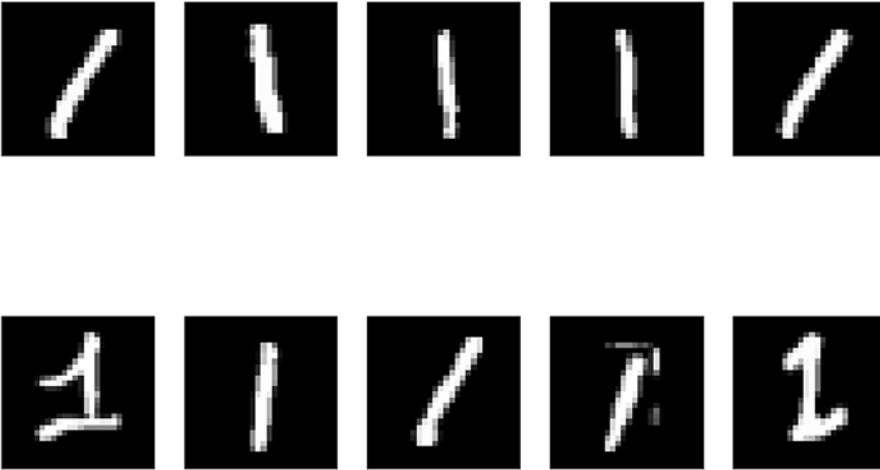


Figure 3-20. The output of running the code in Figure 3-19. Notice the amount of variation, as well as anomalous data that you would barely consider as numbers

Now, extend the shape by a dimension. Right now, the dimensions of the training and testing sets are as shown in Figure 3-21 and Figure 3-22.

```
print("x_train: {}\nx_test: {}".format(
    x_train.shape, x_test.shape, ))
```

Figure 3-21. Code to output the shapes of the training and testing data sets

```
In [71]: 1 print("x_train: {}\nx_test: {}".format(
2         x_train.shape, x_test.shape, ))

x_train: (60000, 28, 28)
x_test: (10000, 28, 28)
```

Figure 3-22. The output of running the code in Figure 3-21

For the purposes of training your model, you want to extend this shape to (60000, 28, 28, 1) and (10000, 28, 28, 1).

A property of images is that there are three dimensions for color images and two for grey scale images. Grey scale images are simply row x column since they don't have color channels. Color images, on the other hand, can be formatted as row x column x channel or channel x row x column. For color images, the variable channel is 3 because you want to know the pixel values for red, green, and blue (RGB).

In this case, it's grey scale, so you don't have to worry about the channel variable, but the following code will account for both cases if you end up using a data set with color such as the CIFAR-10 data set. CIFAR-10 is extremely similar to MNIST, but this time you are classifying the 32x32 images based on labels such as cars, birds, ships, etc. and they are in color. Run the code in Figure 3-23.

```
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, im_row,
                              im_col)
    x_test = x_test.reshape(x_test.shape[0], 1, im_row,
                             im_col)
    input_shape = (1, im_row, im_col)
else:
    x_train = x_train.reshape(x_train.shape[0], im_row,
                              im_col, 1)
    x_test = x_test.reshape(x_test.shape[0], im_row, im_col,
                             1)
    input_shape = (im_row, im_col, 1)
```

Figure 3-23. Code to reshape the training and testing data sets depending on whether or not the channels are first, and then to define the input shape of the model

Now convert the values to float32 and divide by 255. Right now, the values are all integer values that range from 0 to 255, but you want to convert those values to float and make them 0 to 1. This is a process called **normalization**, or **feature scaling**, where you attempt to rescale the data to smaller, more manageable values. In this example, you use a method called **min-max normalization**, defined by the formula in Figure 3-24.

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Figure 3-24. Formula for min-max normalization

Your values ranged from 0 to 255. For each value, you “subtracted” 0 from x , and divided by $255 - 0$, which is just 255. Rescaling the pixel values from a range of $[0, 255]$ to $[0, 1]$ is common in image tasks and can be done with colored images as well.

There are other methods, including **mean normalization**, **standardization (z-score normalization)**, and **unit length scaling**.

The formulas for each method are as follows:

Mean normalization (Figure 3-25)

$$x' = \frac{x - x_{average}}{x_{max} - x_{min}}$$

Figure 3-25. Formula for mean normalization

This formula is similar to min-max normalization, except you use $x_{average}$ in the numerator over x_{min} .

Standardization (Figure 3-26)

$$x' = \frac{x - \bar{x}}{\sigma}$$

Figure 3-26. Formula for standardization

You basically find z-score values for each x and use those instead of the original x values.

Unit length scaling (Figure 3-27)

$$x' = \frac{x}{\|x\|}$$

Figure 3-27. Formula for unit length scaling

You find the unit vector for x and use that instead. Unit vectors have a magnitude of 1. The next block of code is shown in Figure 3-28.

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

y_train = keras.utils.to_categorical(y_train, n_classes)
y_test = keras.utils.to_categorical(y_test, n_classes)
```

Figure 3-28. Converting x_{train} and x_{test} to float32 and applying min-max normalization by dividing by 255. For y_{train} and y_{test} , you convert them to a one-hot encoded format

What `keras.utils.to_categorical()` does is take the vector of classes and create a binary class matrix of the number of classes. Assume that you have a vector representing y_{train} with 6 classes at most, going from 0-5 (Figure 3-29).

Data at index 0	1
Data at index 1	5
Data at index 2	4
Data at index 3	2

Figure 3-29. A vector representing y_{train} that has 6 classes with values ranging from 0-5

After running `keras.utils.to_categorical(y_train, n_classes)` where `n_classes = 5`, Figure 3-30 shows what you would now get for y_{train} .

Data at index 0	0	1	0	0	0	0
Data at index 1	0	0	0	0	0	1
Data at index 2	0	0	0	0	1	0
Data at index 3	0	0	1	0	0	0

Figure 3-30. A one-hot encoded representation of the `y_train` vector in Figure 3-39

The classes are still the same, but this time you have to get the class by their index and not by direct value. At index 1 (row 1 if you think of this as a matrix with 1 column) of the original vector, you see that the class label is 5. In your transformed `y_train` data (which is now a matrix), at row 1 (previously index 1 before the transformation), you see that everything is a 0 in the vector at that index except for the value at column 5. And so, `y_train` is still 5 at index 1, but it's formatted differently.

Now let's check the shapes of your transformed data in Figure 3-31 and Figure 3-32.

```
print("x_train: {}\nx_test: {}\ninput_shape: {}\n# of training
samples: {}\n# of testing samples: {}".format(
x_train.shape, x_test.shape, input_shape, x_train.shape[0],
x_test.shape[0]))
```

Figure 3-31. Print the shapes of the transformed data

```
In [126]: 1 print("x_train: {}\nx_test: {}\ninput_shape: {}\n \
2 # of training samples: {}\n# of testing samples: {}".format(
3 x_train.shape, x_test.shape, input_shape, x_train.shape[0], x_test.shape[0]))

x_train: (60000, 28, 28, 1)
x_test: (10000, 28, 28, 1)
input_shape: (28, 28, 1)
# of training samples: 60000
# of testing samples: 10000
```

Figure 3-32. The resulting output

Note The \ character tells Python that you want to continue to the next line. Without it, the code would not run because Python doesn't see the end of the string denoted by the second ", but what \ tells Python is to continue on the next line.

Now you can move on to defining and compiling your model.

Run the code in Figure 3-33.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3),
                 activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(n_classes,
                 activation='softmax'))

model.compile(loss=keras.losses.categorical_
              crossentropy,

              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])

model.summary()
```

Figure 3-33. Code to define a deep learning model and add layers to it

In Keras, the **sequential** model is a stack of layers. The Conv2D is a two-dimensional convolutional layer.

In convolutional neural networks, a **convolution layer** filters through the data and multiplies each of the values element-wise by the weights in the filter and sums them up to generate one value. In this case, it's a 3x3 filter that slides over each of the pixels to generate a smaller layer called an **activation map** or **feature map**. This feature map then has another filter applied to it in the second convolutional layer to generate another, smaller feature map. The weights that are optimized during backpropagation are found in the filter. To get a better idea of this, let's look at some examples of how this works.

Assume a 5x5 pixel picture like Figure 3-34.

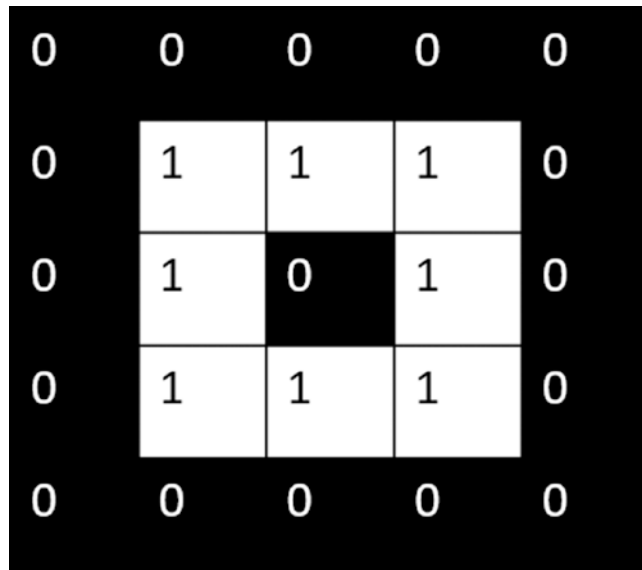


Figure 3-34. A 5x5 pixel picture, with 0 representing black pixels and 1 representing white pixels

Assume also that your kernel size (filter dimensions) is 2x2. Figure 3-35 shows how the convolutions would go.

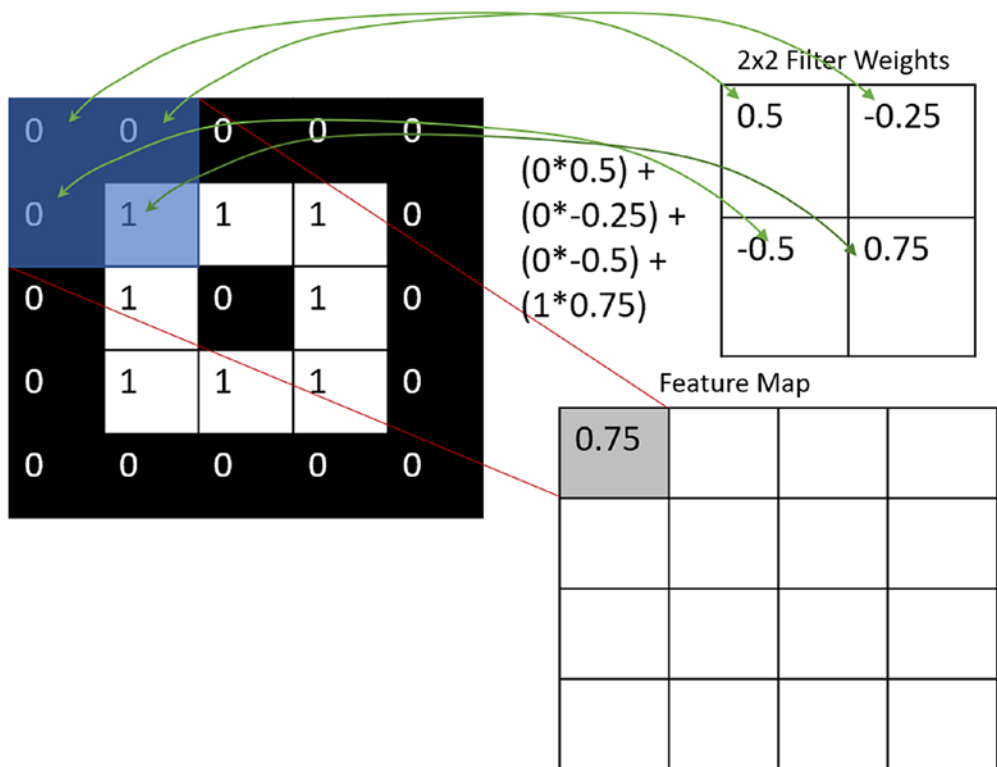


Figure 3-35. An example of one multiplication of the 2x2 filter on a 2x2 section of the input image. The filter weights are applied element-wise and produce an output value that is part of the feature map—the output of this convolutional layer

To begin with, you have a random set of weights for the 2x2 **filter**, or **kernel**. The filter goes over the first 2x2 region in the image and sums the element-wise multiplication of the values in the filter and the values in the 2x2 region of the image. This value is the first element of the feature map, which is a 4x4 layer image. Given an **nxn** filter and **mxm** image, your feature map dimensions will be an **m-n+1 x m-n+1** dimensional image. In this case, your image is 5x5 and the kernel is 2x2, so the feature map is $5 - 2 + 1 = 4 \times 4$ pixels.

The filter goes through each region in the image pixel by pixel, as shown in Figure 3-36.

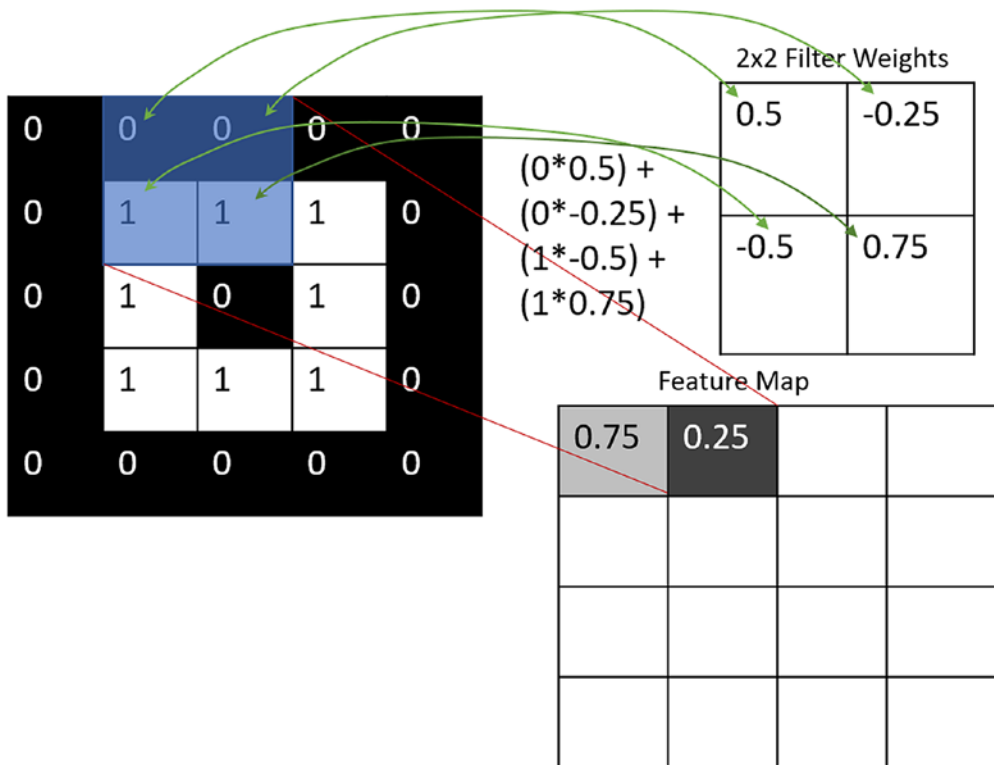


Figure 3-36. After the operation in Figure 3-35, the filter moves to the next set of data to multiply over, producing the second value in the feature map

The filter continues doing this until it reaches the right side of the image. After that, the filter goes one down and starts again from the left side of the image, like in Figure 3-37.

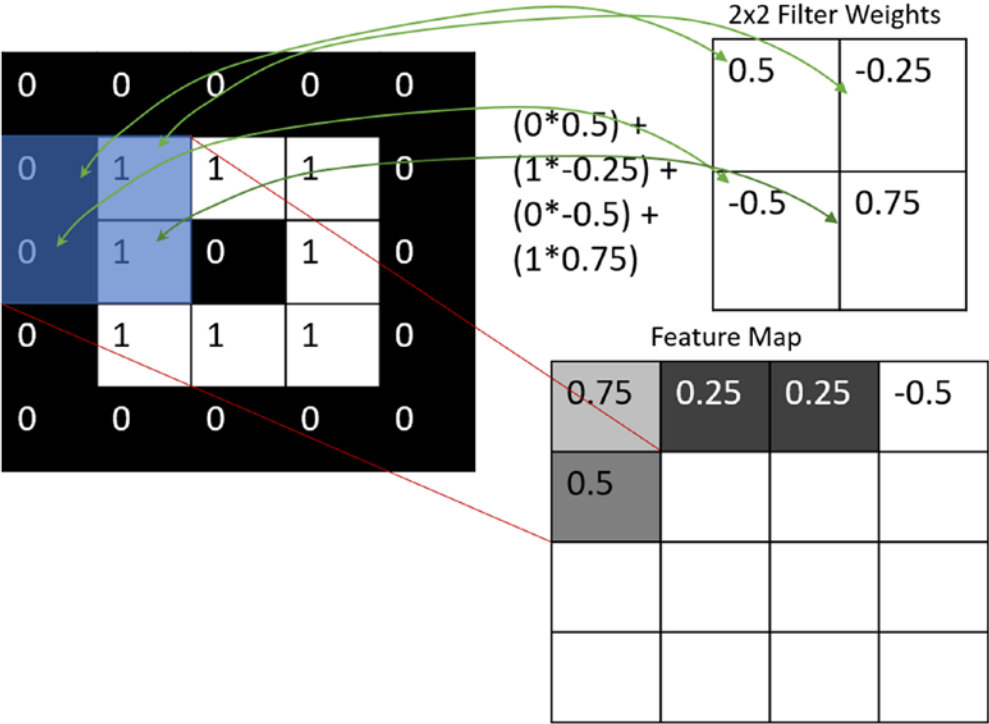


Figure 3-37. Showing what happens after the filter reaches the right-most side of the image. It moves down one (in this case, at least; you can specify how much you want the filter to move as a parameter when calling this layer) and then continues its operations as usual

From here, the filter continues moving right in a pixel by pixel fashion (see Figure 3-38).

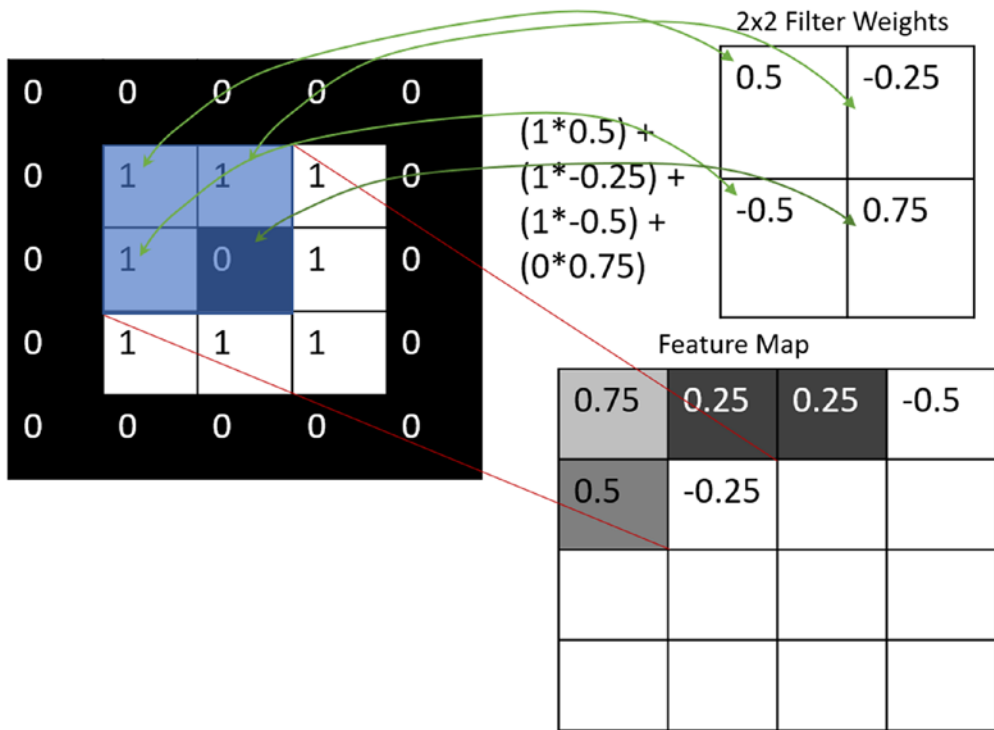


Figure 3-38. The filter continues moving as normal, adding more values to the feature map

Once it reaches the end, it goes back to the first column and down one row and continues its operations until it reaches the bottom right region (see Figure 3-39).

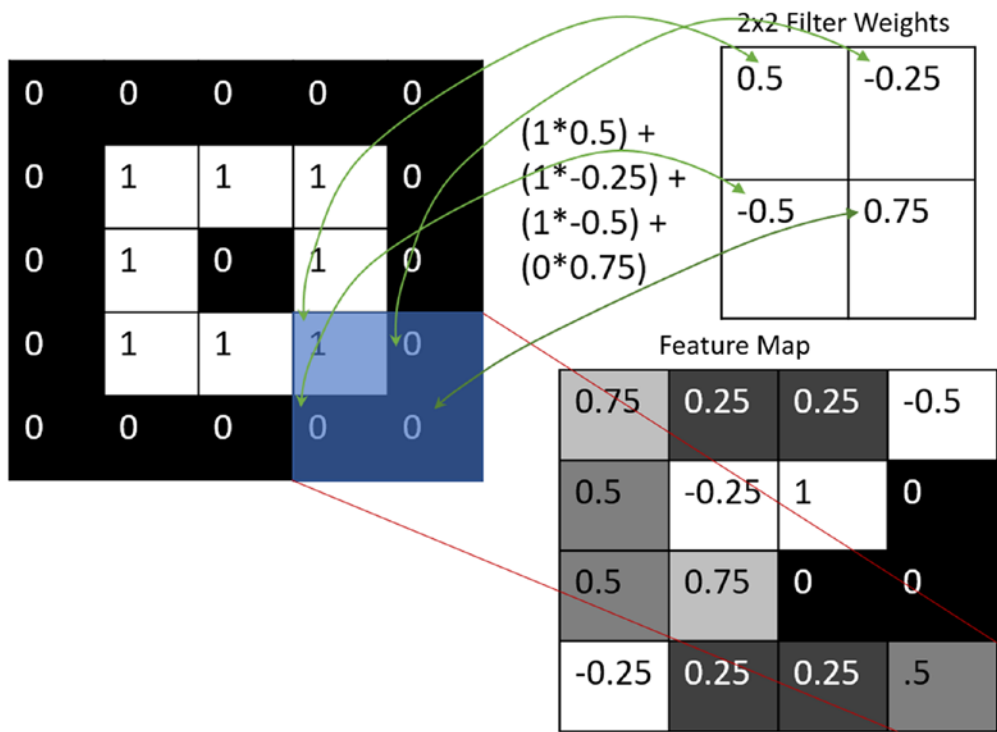


Figure 3-39. Once the filter reaches this value here, the convolution operation ceases, outputting a feature map to the next layer

The feature map doesn't make much sense due to the randomness of the weights.

After the two convolutional layers, you run into the MaxPooling2D layer. **Max pooling** is where the input data is scanned by a filter, which in this case is a 2x2 filter, and the maximum value in the 2x2 region of the image is chosen to be the value in the new n-dimensional image. If the **stride length** is not given, by default Keras chooses the pool size. The stride length is how far the filter should shift, and it plays a role in determining the feature map size. In this case, since the stride length is 2 and the pooling filter size is also 2x2, the dimensions of the input data are reduced in half.

Assume that the 4x4 image in Figure 3-40 is the input to a max pooling layer with pool size of 2x2.

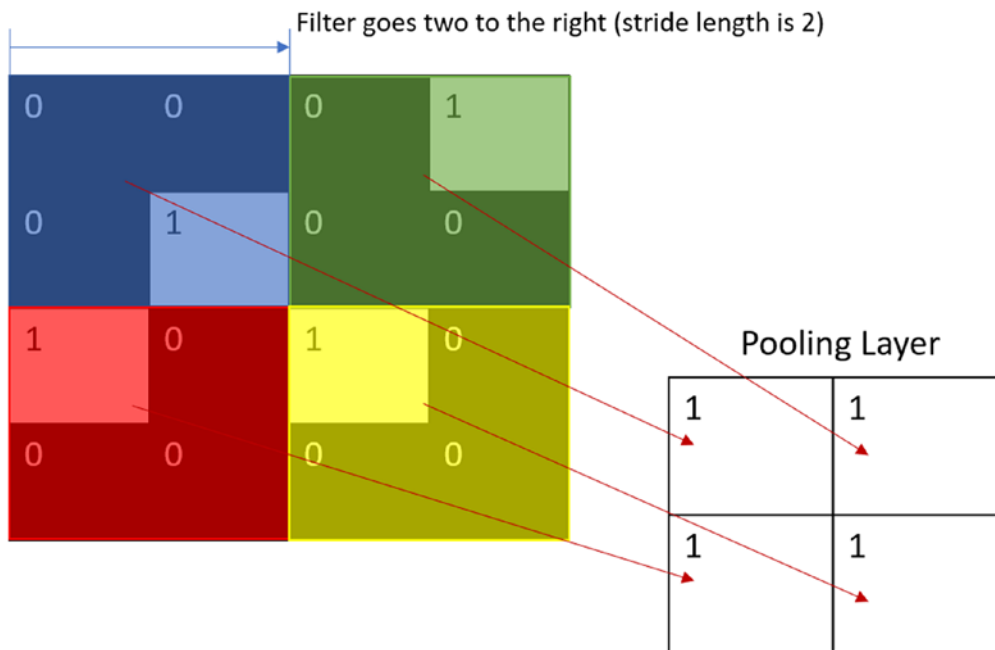


Figure 3-40. What a max pooling operation looks like on a 4x4 image

Since the pool size is 2x2 and the stride length is also 2 in this case (no parameter was provided for stride length), the pooling layer happens to split the entire image into regions of 2x2 pooling filters.

If the stride length was 1, then you would have a situation similar to the convolution example you saw earlier, and the dimensions of the feature map would be $4-2+1 = 3 \times 3$. This process of pooling can also be referred to as **downsampling**.

The **pooling layer** helps reduce the size of the data to allow for easier computation. Additionally, it can help with pattern identification because the maximum value in each region is selected, allowing for the patterns to stand out more.

The **dropout** layer is next. Dropout is a regularization technique where a proportion (this is a parameter passed in) of randomly selected nodes are “dropped,” or ignored during the training process.

Flatten is a layer where the entire input is squashed into one dimension. Assume that you are trying to flatten a 3x3 image, like Figure 3-41.

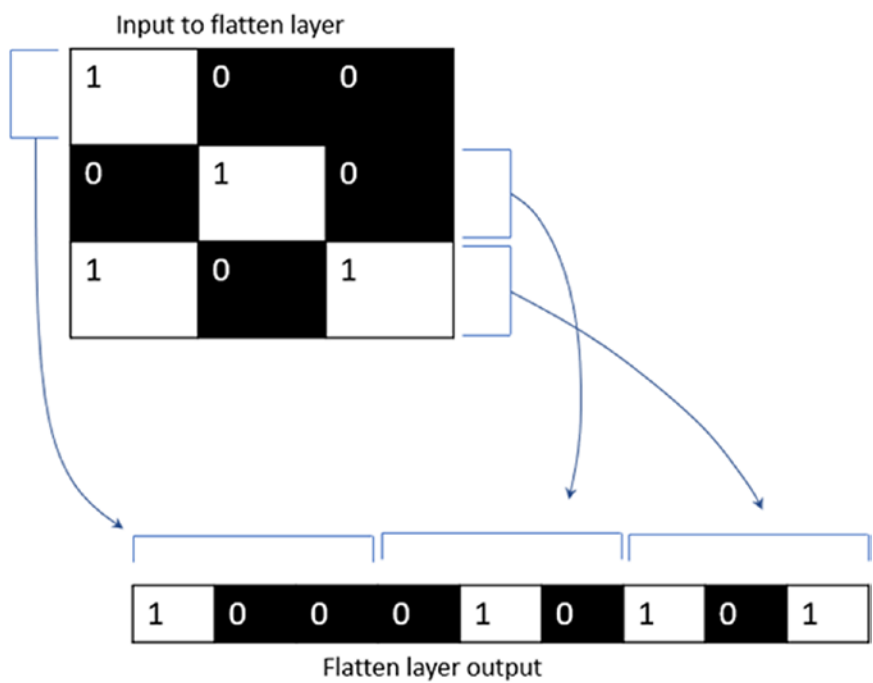


Figure 3-41. Showing what a flatten layer does to an input 3x3 image

The **dense** layer is simply a layer of regular nodes similar to those in the artificial neural network example. They perform in the same way, but in this case the number of nodes varies from 128 in the first dense layer and to 10 in the second dense layer. The activation function also changes, from ‘relu,’ or **rectified linear unit (ReLU)** in the first dense layer, to **softmax** in the second.

Mathematically, the **ReLU** function is defined as $y = \max(0, x)$, so when the node calculates the dot products between the input and the weights and adds the bias, it simply outputs whatever is bigger between 0 or the calculation.

The graph for ReLU looks like Figure 3-42.

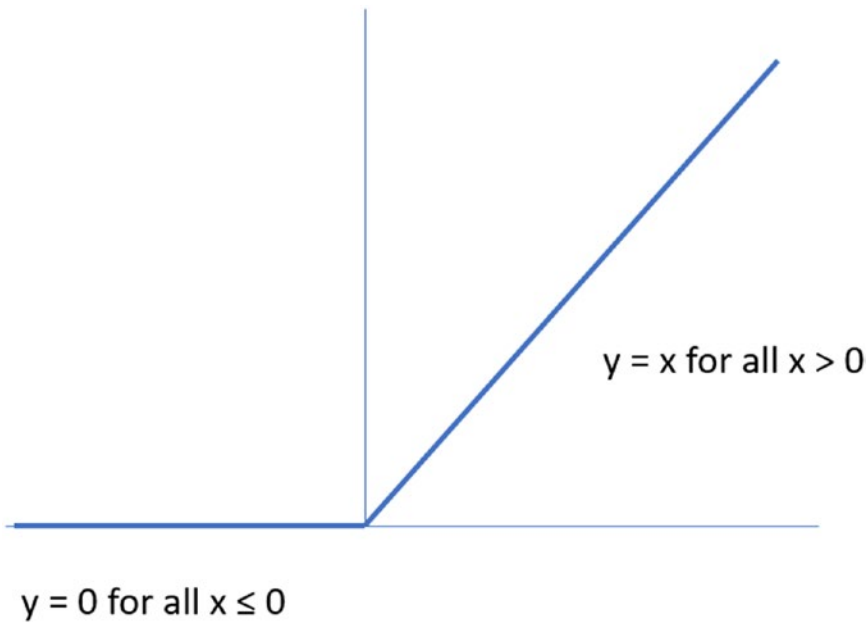


Figure 3-42. A graph showing the ReLU function

The general formula for softmax is shown in Figure 3-43.

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^K e^{x_j}} \quad \text{For } i = 1, \dots, K \text{ and } x = (x_1, \dots, x_k) \in R^k$$

Figure 3-43. Formula for the softmax activation function

As for the **optimizer**, it is set to the **Adam optimizer**, a type of gradient-based optimizer. By default, the parameter known as the **learning rate** is set to 0.001. Recall that the learning rate helps determine the step size taken by the optimization algorithm to see how much to adjust the weights by.

After executing the code in Figure 3-43, you get the output in Figure 3-44.

```
In [128]: 1 model = Sequential()
2 model.add(Conv2D(32, kernel_size=(3, 3),
3               activation='relu',
4               input_shape=input_shape))
5 model.add(Conv2D(64, (3, 3), activation='relu'))
6 model.add(MaxPooling2D(pool_size=(2, 2)))
7 model.add(Dropout(0.25))
8 model.add(Flatten())
9 model.add(Dense(128, activation='relu'))
10 model.add(Dropout(0.5))
11 model.add(Dense(n_classes, activation='softmax'))
12
13 model.compile(loss=keras.losses.categorical_crossentropy,
14               optimizer=keras.optimizers.Adam(),
15               metrics=['accuracy'])
16
17 model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_17 (Conv2D)	(None, 26, 26, 32)	320
conv2d_18 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_6 (MaxPooling2D)	(None, 12, 12, 64)	0
dropout_11 (Dropout)	(None, 12, 12, 64)	0
flatten_6 (Flatten)	(None, 9216)	0
dense_10 (Dense)	(None, 128)	1179776
dropout_12 (Dropout)	(None, 128)	0
dense_11 (Dense)	(None, 10)	1290
Total params: 1,199,882		
Trainable params: 1,199,882		
Non-trainable params: 0		

Figure 3-44. The output for the code in Figure 3-33. Note how it tells you the output shapes of each layer and the number of parameters; this can be useful when creating custom models and finding out that there is a mismatch between the dimensionality of what a layer expects and what it actually receives

Now let's move on to training the data. Depending on your setup, this can take anywhere from a few seconds to several minutes. Without cuda, expect that this will take much longer.

Run the code in Figure 3-45.

```
checkpoint = ModelCheckpoint(filepath="keras_MNIST_CNN.h5",  
                             verbose=0,  
                             save_best_only=True)  
  
model.fit(x_train, y_train,  
          batch_size=batch_size,  
          epochs=n_epochs,  
          verbose=1,  
          validation_data=(x_test, y_test),  
          callbacks=[checkpoint])  
  
score = model.evaluate(x_test, y_test, verbose=0)  
print('Test loss:', score[0])  
print('Test accuracy:', score[1])
```

Figure 3-45. Code to train the model and print accuracy and loss values for the test set

The variable `checkpoint` will store the model in the same folder as this code with the name `keras_MNIST_CNN.h5`. If you don't want to save the model, run the code in Figure 3-46 instead.

```
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=n_epochs,
          verbose=1,
          validation_data=(x_test, y_test))

score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Figure 3-46. Run this code if you don't want to save the model

If successful, you should see something like Figure 3-47.

```
In [265]: 1 checkpoint = ModelCheckpoint(filepath="keras_MNIST_CNN.h5",
2         verbose=0,
3         save_best_only=True)
4
5 model.fit(x_train, y_train,
6         batch_size=batch_size,
7         epochs=n_epochs,
8         verbose=1,
9         validation_data=(x_test, y_test),
10        callbacks=[checkpoint])
11
12 score = model.evaluate(x_test, y_test, verbose=0)
13 print('Test loss:', score[0])
14 print('Test accuracy:', score[1])
```

Train on 60000 samples, validate on 10000 samples

```
Epoch 1/15
60000/60000 [=====] - 9s - loss: 0.2444 - acc: 0.9258 - val_loss: 0.0485 - val_acc: 0.9844
Epoch 2/15
60000/60000 [=====] - 7s - loss: 0.0878 - acc: 0.9743 - val_loss: 0.0391 - val_acc: 0.9869
Epoch 3/15
60000/60000 [=====] - 7s - loss: 0.0625 - acc: 0.9815 - val_loss: 0.0330 - val_acc: 0.9893
Epoch 4/15
60000/60000 [=====] - 7s - loss: 0.0531 - acc: 0.9844 - val_loss: 0.0322 - val_acc: 0.9898
Epoch 5/15
60000/60000 [=====] - 7s - loss: 0.0449 - acc: 0.9865 - val_loss: 0.0329 - val_acc: 0.9889
Epoch 6/15
60000/60000 [=====] - 7s - loss: 0.0410 - acc: 0.9875 - val_loss: 0.0281 - val_acc: 0.9914
Epoch 7/15
60000/60000 [=====] - 7s - loss: 0.0345 - acc: 0.9889 - val_loss: 0.0283 - val_acc: 0.9910
Epoch 8/15
60000/60000 [=====] - 7s - loss: 0.0314 - acc: 0.9902 - val_loss: 0.0265 - val_acc: 0.9926
Epoch 9/15
60000/60000 [=====] - 7s - loss: 0.0293 - acc: 0.9907 - val_loss: 0.0280 - val_acc: 0.9920
Epoch 10/15
60000/60000 [=====] - 7s - loss: 0.0256 - acc: 0.9917 - val_loss: 0.0290 - val_acc: 0.9917
Epoch 11/15
60000/60000 [=====] - 7s - loss: 0.0234 - acc: 0.9927 - val_loss: 0.0269 - val_acc: 0.99260.9
Epoch 12/15
60000/60000 [=====] - 7s - loss: 0.0230 - acc: 0.9925 - val_loss: 0.0267 - val_acc: 0.9923
Epoch 13/15
60000/60000 [=====] - 7s - loss: 0.0197 - acc: 0.9933 - val_loss: 0.0299 - val_acc: 0.9916
Epoch 14/15
60000/60000 [=====] - 7s - loss: 0.0202 - acc: 0.9931 - val_loss: 0.0288 - val_acc: 0.9932
Epoch 15/15
60000/60000 [=====] - 7s - loss: 0.0179 - acc: 0.9938 - val_loss: 0.0261 - val_acc: 0.9934
Test loss: 0.026139157172246224
Test accuracy: 0.9934
```

Figure 3-47. The output of running the training function, accompanied by the loss and accuracy values for the test set

Let's check the AUC score for this. Run the code in Figure 3-48.

```

from sklearn.metrics import roc_auc_score

preds = model.predict(x_test)
auc = roc_auc_score(np.round(preds), y_test)
print("AUC: {:.2%}".format(auc))

```

Figure 3-48. Code to generate the AUC score for this model based on the test set

Basically, the variable predictions are a list of arrays with 10 elements, each containing the probability values for class predictions for each of the `x_test` data samples.

To check the values for the predictions before doing `np.round()`, run the code in Figure 3-49 and see the results in Figure 3-50.

```

preds = model.predict(x_test)
print("Predictions for x_test[0]: {}\n\nActual label for x_test[0]:  
{}\n".format(preds[0], y_test[0]))
print("Predictions for x_test[0] after rounding:  
{}\n".format(np.round(preds)[0]))

```

Figure 3-49. Code to see what the predictions actually look like before rounding them

```
In [267]: 1 preds = model.predict(x_test)
2          print("Predictions for x_test[0]: {}".format(preds[0], y_test[0]))
3          print("Predictions for x_test[0] after rounding: {}".format(np.round(preds)[0]))
4          5

Predictions for x_test[0]: [4.1195924e-19 4.8884741e-14 1.1587565e-13 1.5126733e-13 1.3377293e-15
7.9817291e-17 2.9398691e-23 1.0000000e+00 5.9718682e-15 1.5278325e-13]

Actual label for x_test[0]: [0. 0. 0. 0. 0. 0. 1. 0. 0.]

Predictions for x_test[0] after rounding: [0. 0. 0. 0. 0. 0. 1. 0. 0.]
```

Figure 3-50. The output for running the code in Figure 3-49

The data values for the predictions for every other class besides the one it predicts correctly are so small that rounding them off is insignificant. The AUC score is shown in Figure 3-51.

```
In [266]: 1 from sklearn.metrics import roc_auc_score
2
3 preds = model.predict(x_test)
4 auc = roc_auc_score(np.round(preds), y_test)
5 print("AUC: {:.2%}".format (auc))

AUC: 99.64%
```

Figure 3-51. The generated AUC score for the model. This is the output of running the code in Figure 3-48

That's a really good AUC score! This score indicates that this model is really good at identifying handwritten digits, provided they're in a similar format to the MNIST data set you used during training.

Referring back to the convolutional layers, let's run some code to see what the feature maps look like after the first two convolutional layers compared to the original image.

Run the code in Figure 3-52 and look at the output in Figure 3-53.

```

from keras import models

layers = [layer.output for layer in model.layers[:4]]
model_layers = models.Model(inputs=model.input, outputs=layers)
activations = model_layers.predict(x_train)

fig = plt.figure(figsize=(15,10))

plt.subplot(1, 3, 1)
plt.title("Original")
plt.imshow(x_train[7].reshape(28, 28), cmap='gray')
plt.xticks([])
plt.yticks([])

for f in range(1, 3):
    plt.subplot(1, 3, f+1)
    plt.title("Convolutional layer %d" % f)
    layer_activation = activations[f]
    plt.imshow(layer_activation[7, :, :, 0], cmap='gray')
    plt.xticks([])
    plt.yticks([])

plt.show()

```

Figure 3-52. Code to generate graphs of what the images look like at various stages of the model

```

In [289]: 1 from keras import models
          2
          3 layers = [layer.output for layer in model.layers[:4]]
          4 model_layers = models.Model(inputs=model.input, outputs=layers)
          5 activations = model_layers.predict(x_train)
          6
          7 fig = plt.figure(figsize=(15,10))
          8
          9 plt.subplot(1, 3, 1)
         10 plt.title("Original")
         11 plt.imshow(x_train[7].reshape(28, 28), cmap='gray')
         12 plt.xticks([])
         13 plt.yticks([])
         14
         15 for f in range(1, 3):
         16     plt.subplot(1, 3, f+1)
         17     plt.title("Convolutional layer %d" % f)
         18     layer_activation = activations[f]
         19     plt.imshow(layer_activation[7, :, :, 0], cmap='gray')
         20     plt.xticks([])
         21     plt.yticks([])
         22
         23 plt.show()
         24

```

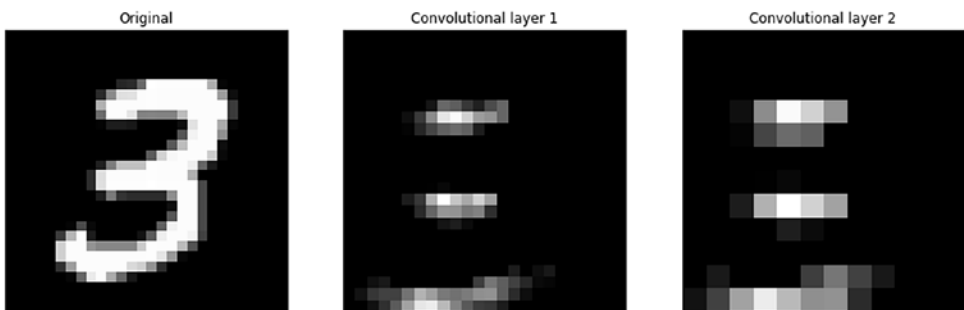


Figure 3-53. The output of running the code in Figure 3-52

As the image passes through the convolutional layers, its dimensions get reduced and the patterns become more apparent. While to us that might not look so much like a three, the model identifies those patterns from the original image and bases its prediction on that.

So now you have a much better understanding of what a CNN is and how Keras can be used to easily create and train your very own deep neural network. If you would like to explore the framework further, feel free to check out Appendix A. If you have any further questions, or would like to explore Keras beyond what's in Appendix A, check out the official Keras documentation.

Intro to PyTorch: A Simple Classifier Model

Now that you have a better idea of what a CNN is and how a classifier model looks like in Keras, let's jump straight into implementing a CNN in PyTorch.

PyTorch doesn't abstract everything to the extent that Keras does, so there's a bit more syntax involved. If you would like to explore this framework further, check out Appendix B, where we cover the basics of PyTorch, its functionality, and apply it to the models that you will explore in Chapter 7.

Just like in Keras, however, you start by importing the necessary modules and defining your hyperparameters (Figure 3-54).

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
import torch.nn.functional as F
import numpy as np

#If cuda device exists, use that. If not, default to CPU.
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

Figure 3-54. Code to import the modules you need and to define the device (CPU or GPU) to run PyTorch on

In PyTorch, you must specify to torch that you want to use the GPU if it exists. In Keras, since you are using tensorflow-gpu as the back end (what Keras runs on top of), it is expected that you have a GPU, CUDA, and cuDNN installed.

Now configure your hyperparameters (Figure 3-55).


```
#Hyperparameters
num_epochs = 15
num_classes = 10
batch_size = 128
learning_rate = 0.001
```

Figure 3-55. Code to define the hyperparameters to use

In this example, you will match the model architecture used in the example for Keras as best as PyTorch allows you to. Not every function is equivalent between TensorFlow and PyTorch, but the vast majority of them are.

Now create your testing and training data sets (Figure 3-56).

```
#Load MNIST data set
train_dataset = torchvision.datasets.MNIST(root='.././data/',
                                           train=True,
                                           transform=transforms.ToTensor(),
                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='.././data/',
                                           train=False,
                                           transform=transforms.ToTensor())

#Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                           batch_size=batch_size,
                                           shuffle=False)
```

Figure 3-56. Using DataLoaders, a feature of PyTorch, to get the training and testing data

The procedure for loading the MNIST data might be a bit different in PyTorch, using data loaders instead of data frames, but you can still use data frames, arrays, and so on in PyTorch after converting them to tensors. The procedure is usually to convert the data frame to a numpy array and then to a PyTorch tensor.

Let's move on to creating your model (Figure 3-57).

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dense1 = nn.Linear(12*12*64, 128)
        self.dense2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.dropout(x, 0.25)
        x = x.view(-1, 12*12*64)
        x = F.relu(self.dense1(x))
        x = F.dropout(x, 0.5)
        x = self.dense2(x)
        return F.log_softmax(x, dim=1)
```

Figure 3-57. *Creation of a convolutional neural network in PyTorch*

The procedure is a bit different than in Keras. In this example, the major layers were defined under `__init__`, which are your two convolutional layers and the two dense layers. The rest of the layers are defined under `forward()`. In `forward()`, you set `x` equal to the output of the activation function of the first convolutional layer. This new `x` is now the input of the next convolutional layer, and you set `x` equal to the output of the activation function of the second convolution layer. This same process repeats for the other layers, but the exact flow of data can be a bit confusing, so Figure 3-58 shows an example of what this code actually does.

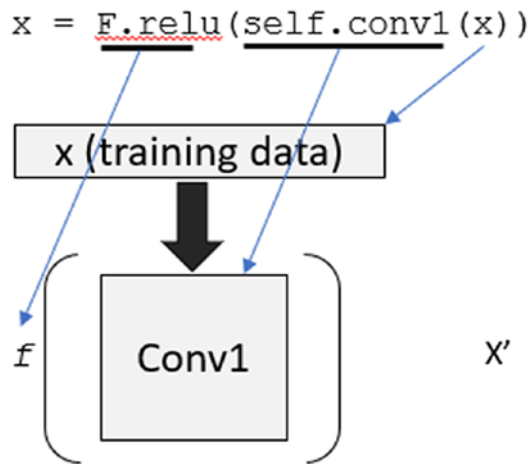


Figure 3-58. *F.relu* is $f(x)$, x is the training data, and *self.conv1* is the first convolutional layer

The original inputs of `x`, `self.conv1`, and `F.relu` can be shown as such. `x` passes into the convolutional layer, and the outputs of that layer pass through the ReLU function. Then you get your final output `X'` (Figure 3-59).

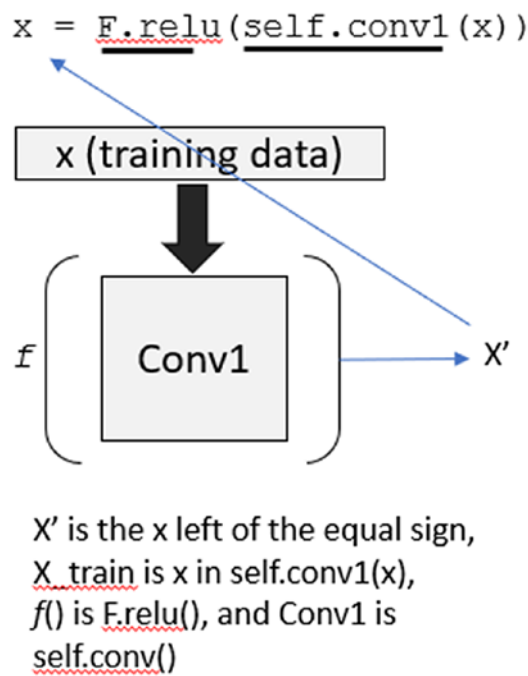


Figure 3-59. The outputs of $f(x)$ are now the new x . Basically, $x = f(x)$. In this case, the output x' is the new x

Now, X is X' , and this new X gets passed onto the next layer (Figure 3-60).

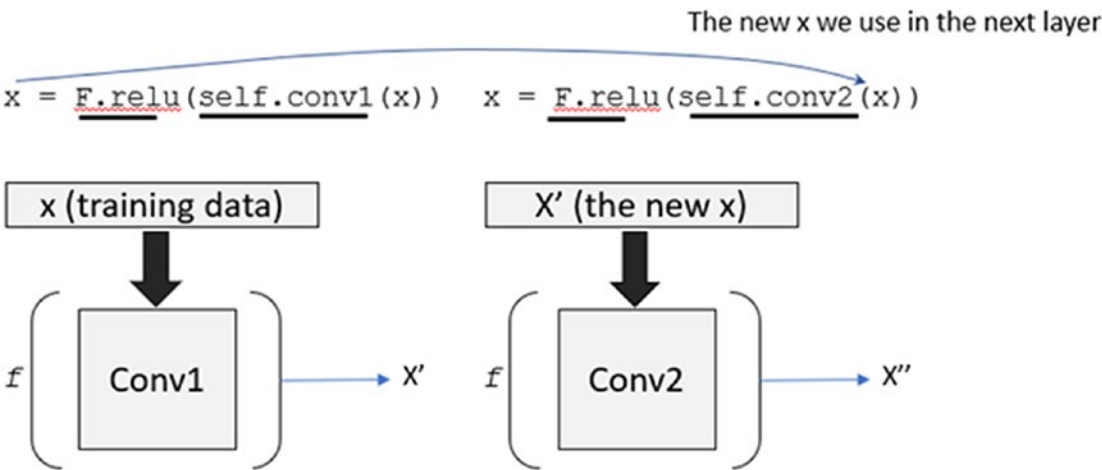


Figure 3-60. The new x is now the new input for the next convolutional layer

The same process repeats again, except with the new value of x (Figure 3-61).

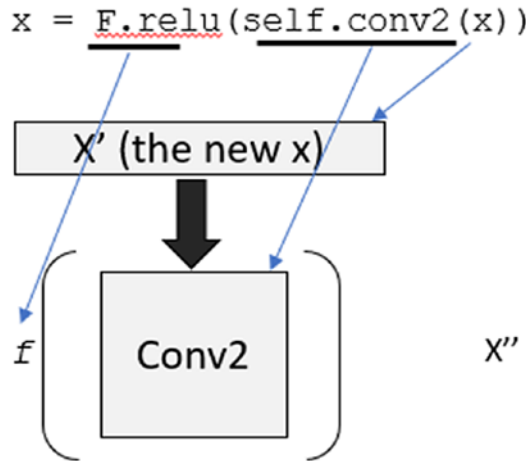


Figure 3-61. The same process repeats, leading to a new value for x

And now you get the new output x' (Figure 3-62).

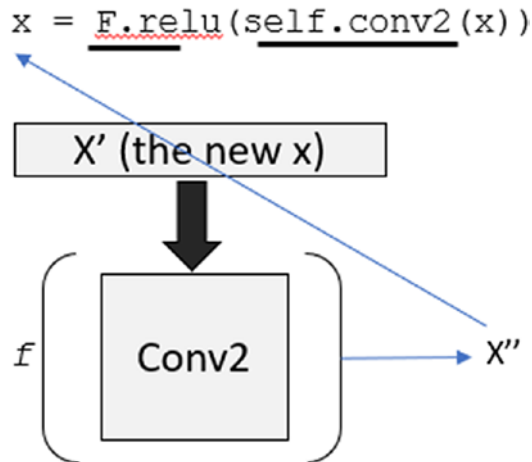


Figure 3-62. Once again, you redefine x as x'' . The process then continues for as many layers as you have in the network

This new output x'' is then the new value of x , and the process continues.

This is the same logic behind the rest of the code, where the output of the activation layer for the old is now the new definition of `x`. This new `x` then goes to the next layer, where a function is applied after it goes through a layer and then that data becomes the new definition of `x`, and so on.

So

```
x = x.view(-1, 12*12*64)
```

performs the same function as the `flatten` layer in the Keras example.

Now you can move on to training your data (Figure 3-63).

```
model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if (i+1) % 100 == 0:
        print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
              .format(epoch+1, num_epochs, i+1, total_step,
                      loss.item()))
```

Figure 3-63. You initialize the model, your loss function, and your optimizer, and then you start the training process

It might take a while, but you should see something like Figure 3-64.

```
In [85]: 1 model = CNN().to(device)
2 criterion = nn.CrossEntropyLoss()
3 optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
4
5 total_step = len(train_loader)
6 for epoch in range(num_epochs):
7     for i, (images, labels) in enumerate(train_loader):
8         images = images.to(device)
9         labels = labels.to(device)
10
11         # Forward pass
12         outputs = model(images)
13         loss = criterion(outputs, labels)
14
15         # Backward and optimize
16         optimizer.zero_grad()
17         loss.backward()
18         optimizer.step()
19
20     if (i+1) % 100 == 0:
21         print ('Epoch [{} / {}], Step [{} / {}], Loss: {:.4f}'
22               .format(epoch+1, num_epochs, i+1, total_step, loss.item()))
23
```

```
Epoch [1/15], Step [100/469], Loss: 0.1666
Epoch [1/15], Step [200/469], Loss: 0.2753
Epoch [1/15], Step [300/469], Loss: 0.2462
Epoch [1/15], Step [400/469], Loss: 0.1169
Epoch [2/15], Step [100/469], Loss: 0.0327
Epoch [2/15], Step [200/469], Loss: 0.0238
Epoch [2/15], Step [300/469], Loss: 0.0293
Epoch [2/15], Step [400/469], Loss: 0.0598
Epoch [3/15], Step [100/469], Loss: 0.0179
Epoch [3/15], Step [200/469], Loss: 0.0577
Epoch [3/15], Step [300/469], Loss: 0.0275
Epoch [3/15], Step [400/469], Loss: 0.0228
Epoch [4/15], Step [100/469], Loss: 0.0051
Epoch [4/15], Step [200/469], Loss: 0.0139
Epoch [4/15], Step [300/469], Loss: 0.0048
Epoch [4/15], Step [400/469], Loss: 0.0033
Epoch [5/15], Step [100/469], Loss: 0.0081
Epoch [5/15], Step [200/469], Loss: 0.0044
Epoch [5/15], Step [300/469], Loss: 0.0084
Epoch [5/15], Step [400/469], Loss: 0.0011
Epoch [6/15], Step [100/469], Loss: 0.0077
.....
```

Figure 3-64. The output of the training process

After training is done, you can test your model and find the AUC score (Figure 3-65).

```

from sklearn.metrics import roc_auc_score

preds = []
y_true = []

# Test the model
model.eval() # Set model to evaluation mode.
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        detached_pred = predicted.detach().cpu().numpy()
        detached_label = labels.detach().cpu().numpy()
        for f in range(0, len(detached_pred)):
            preds.append(detached_pred[f])
            y_true.append(detached_label[f])

print('Test Accuracy of the model on the 10000 test images:
{: .2%}'.format(correct / total))

preds = np.eye(num_classes)[preds]
y_true = np.eye(num_classes)[y_true]
auc = roc_auc_score(preds, y_true)
print("AUC: {: .2%}".format (auc))

# Save the model checkpoint
torch.save(model.state_dict(), 'pytorch_mnist_cnn.ckpt')

```

Figure 3-65. Code to evaluate the model and generate the AUC score

The resulting output is shown in Figure 3-66.

```
Epoch [15/15], Step [300/469], Loss: 0.0131
Epoch [15/15], Step [400/469], Loss: 0.0002

In [87]: 1 from sklearn.metrics import roc_auc_score
2
3 preds = []
4 y_true = []
5 # Test the model
6 model.eval() # Set model to evaluation mode.
7 with torch.no_grad():
8     correct = 0
9     total = 0
10    for images, labels in test_loader:
11        images = images.to(device)
12        labels = labels.to(device)
13        outputs = model(images)
14        _, predicted = torch.max(outputs.data, 1)
15        total += labels.size(0)
16        correct += (predicted == labels).sum().item()
17        detached_pred = predicted.detach().cpu().numpy()
18        detached_label = labels.detach().cpu().numpy()
19        for f in range(0, len(detached_pred)):
20            preds.append(detached_pred[f])
21            y_true.append(detached_label[f])
22
23    print('Test Accuracy of the model on the 10000 test images: {:.2%}'.format(correct / total))
24
25    preds = np.eye(num_classes)[preds]
26    y_true = np.eye(num_classes)[y_true]
27    auc = roc_auc_score(preds, y_true)
28    print("AUC: {:.2%}".format (auc))
29 # Save the model checkpoint
30 torch.save(model.state_dict(), 'pytorch_mnist_cnn.ckpt')

Test Accuracy of the model on the 10000 test images: 99.07%
AUC: 99.48%
```

Figure 3-66. The generated accuracy score on the test set and the AUC score for the model

Now you a bit more about how to create and train your own CNN in PyTorch. PyTorch is a bit harder to learn than Keras, which aims to make everything quite readable and simple, having abstracted all of the more complicated bits of code. TensorFlow and PyTorch are both low-level APIs that require more code to be written because of the lack of abstraction, but offer more flexibility in controlling exactly how you want everything to be. Between the two, PyTorch is easier to debug if you're using the debugging tool in PyCharm. In the end, it's all a matter of preference, although TensorFlow and PyTorch both perform faster on larger data sets.

If you would like to explore PyTorch further, check out Appendix B, where we cover a more refined way to create models, train, and test, as well as the general functionality that PyTorch offers. Appendix B also applies PyTorch to the models in Chapter 7, which are done in Keras.

If you would like to learn more about PyTorch after visiting Appendix B, check out the official PyTorch documentation.

Summary

In recent years, deep learning has revolutionized an incredible variety of fields. Thanks to deep learning, we now have self-driving cars, models that have beaten professionals in detecting certain cancers, instant translation between languages, etc. It is of no surprise, then, that deep learning has also contributed heavily to the field of anomaly detection.

In this chapter, we discussed what deep learning is and what an artificial neural network is. You explored two popular frameworks, Keras and PyTorch, by applying them to the task of image classification with the MNIST data set.

In the upcoming chapters, we will take a look at the applications to anomaly detection of the following types of deep learning models: **autoencoders**, **restricted Boltzmann machines**, **RNN/LSTM** networks, and **temporal convolutional networks**.

In the next chapter, we will look at **unsupervised anomaly detection** with **autoencoders**.

CHAPTER 4

Autoencoders

In this chapter, you will learn about autoencoder neural networks and the different types of autoencoders. You will also learn how autoencoders can be used to detect anomalies and how you can implement anomaly detection using autoencoders.

In a nutshell, the following topics will be covered throughout this chapter:

- What are autoencoders?
- Simple autoencoders
- Sparse autoencoders
- Deep autoencoders
- Convolutional autoencoders
- Denoising autoencoders
- Variational autoencoders

What Are Autoencoders?

In the previous chapter, you learned about the basic functioning of a neural network. The basic concept is that a neural network essentially computes a weighted calculation of inputs to produce outputs. The inputs are in the input layer and the outputs are in the output layer and there are one or more hidden layers between the input and output layers. Back propagation is a technique used to train the network while trying to adjust the weights until the error is minimized. Autoencoders use this property of a neural network in a special way to accomplish some very efficient methods of training networks to learn normal behavior, thus helping to detect anomalies when they occur. Figure 4-1 shows a typical neural network.

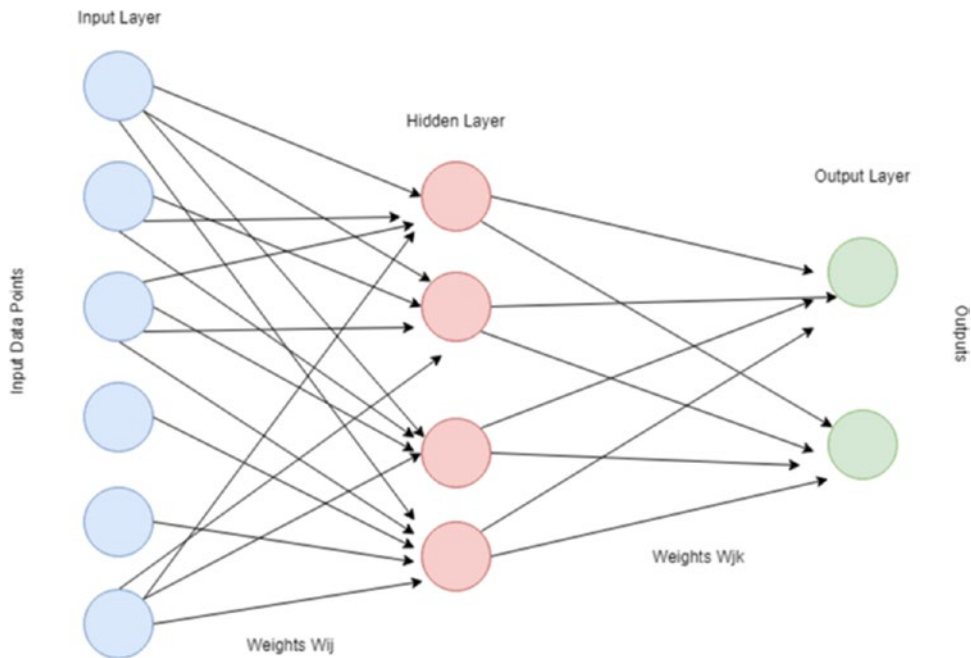


Figure 4-1. A typical neural network

Autoencoders are neural networks that have the ability to discover low-dimensional representations of high-dimensional data and are able to reconstruct the input from the output. Autoencoders are made up of two pieces of the neural network, an encoder and a decoder. The encoder reduces the dimensionality of a high dimensional dataset to a low dimensional one whereas a decoder essentially expands the low-dimensional data to high-dimensional data. The goal of such a process is to try to reconstruct the original input. If the neural network is good, then there is a good chance of reconstructing the original input from the encoded data. This inherent principle is critical in building an anomaly detection module.

Note that autoencoders are not that great if you have training samples containing few dimensions/features at each input point. Autoencoders perform well for five or more dimensions. If you have just one dimension/feature then, as you can imagine, you are just doing a linear transformation, which is not useful.

Autoencoders are incredibly useful in many use cases. Some popular applications of autoencoders are

1. Training deep learning networks
2. Compression
3. Classification
4. Anomaly detection
5. Generative models

Simple Autoencoders

Of course, we will focus on the anomaly detection piece in this chapter. Now, an autoencoder neural network is actually a pair of two connected sub-networks, an encoder and a decoder. An encoder network takes in an input and converts it into a smaller, dense representation, also known as a latent representation of the input, which the decoder network can then use to convert it back to the original input as much as possible. Figure 4-2 shows an example of an autoencoder with encoder and decoder sub-networks.

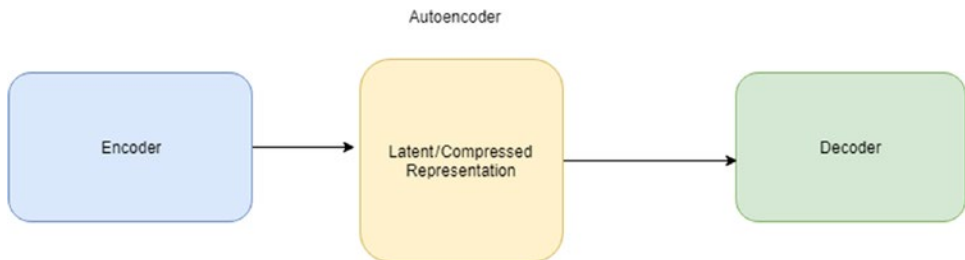


Figure 4-2. A depiction of an autoencoder

Autoencoders use data compression logic where the compression and decompression functions implemented by the neural networks are lossy and are mostly unsupervised without much intervention. Figure 4-3 shows an expanded view of an autoencoder.

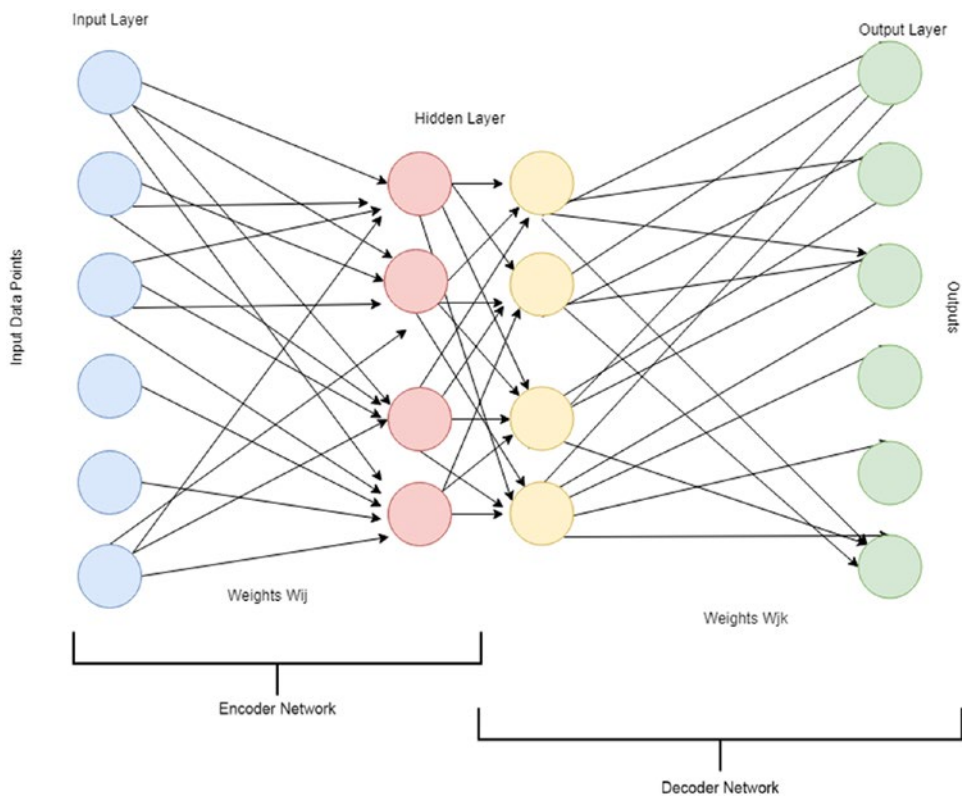


Figure 4-3. Expanded view of an autoencoder

The entire network is usually trained as a whole. The loss function is usually either the mean-squared error or cross-entropy between the output and the input, known as the *reconstruction loss*, which penalizes the network for creating outputs different from the input. Since the encoding (which is simply the output of the hidden layer in the middle) has far less units than the input, the encoder must choose to discard information. The encoder learns to preserve as much of the relevant information as possible in the limited encoding and intelligently discards the irrelevant parts. The decoder learns to take the encoding and properly reconstruct it back into the input. If you are processing images, then the output is an image. If the input is an audio file, the output is an audio file. If the input is some feature engineered dataset, the output will be a dataset too. We will use a credit card transaction sample to illustrate autoencoders in this chapter.

Why do we even bother learning the presentation of the original input only to reconstruct the output as well as possible? The answer is that when we have input with many features, generating a compressed representation via the hidden layers of the neural network could help in compressing the input of the training sample. So when the neural network goes through all the training data and fine tunes the weights of all the hidden layer nodes, what will happen is that the weights will truly represent the kind of input that we typically see. As a result of this, if we try to input some other type of data, such as having data with some noise, the autoencoder network will be able to detect the noise and remove at least some portion of the noise when generating the output. This is truly fantastic because now we can potentially remove noise from, for example, images of cats and dogs. Another example is when security monitoring cameras capture hazy unclear pictures, maybe in the dark or during adverse weather, causing noisy images.

The logic behind the denoising autoencoder is that if we have trained our encoder on good, normal images and the noise when it comes as part of the input is not really a salient characteristic, it is possible to detect and remove such noise.

Figure 4-4 shows the basic code to import all necessary packages in a Jupyter notebook. Note the versions of the various packages.

```

import keras
from keras import optimizers
from keras import losses
from keras.models import Sequential, Model
from keras.layers import Dense, Input, Dropout, Embedding, LSTM
from keras.optimizers import RMSprop, Adam, Nadam
from keras.preprocessing import sequence
from keras.callbacks import TensorBoard

import sklearn
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, roc_auc_score
from sklearn.preprocessing import MinMaxScaler

import seaborn as sns
import pandas as pd
import numpy as np
import matplotlib

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
%matplotlib inline

import tensorflow
import sys
print("Python: ", sys.version)

print("pandas: ", pd.__version__)
print("numpy: ", np.__version__)
print("seaborn: ", sns.__version__)
print("matplotlib: ", matplotlib.__version__)
print("sklearn: ", sklearn.__version__)
print("Keras: ", keras.__version__)
print("Tensorflow: ", tensorflow.__version__)

```

Using TensorFlow backend.

```

Python: 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)]
pandas: 0.24.2
numpy: 1.16.3
seaborn: 0.9.0
matplotlib: 3.0.3
sklearn: 0.20.3
Keras: 2.2.4
Tensorflow: 1.13.1

```

Figure 4-4. *Importing packages in a Jupyter notebook*

Figure 4-5 shows the code to visualize the results via a confusion matrix, a chart for the anomalies and a chart for the errors (the difference between predicted and truth) while training. It shows the Visualization helper class.


```

class Visualization:
    labels = ["Normal", "Anomaly"]

    def draw_confusion_matrix(self, y, ypred):
        matrix = confusion_matrix(y, ypred)

        plt.figure(figsize=(10, 8))
        colors = ["orange", "green"]
        sns.heatmap(matrix, xticklabels=self.labels, yticklabels=self.labels, cmap=colors, annot=True, fmt="d")
        plt.title("Confusion Matrix")
        plt.ylabel('Actual')
        plt.xlabel('Predicted')
        plt.show()

    def draw_anomaly(self, y, error, threshold):
        groupsDF = pd.DataFrame({'error': error,
                                'true': y}).groupby('true')

        figure, axes = plt.subplots(figsize=(12, 8))

        for name, group in groupsDF:
            axes.plot(group.index, group.error, marker='x' if name == 1 else 'o', linestyle='',
                      color='r' if name == 1 else 'g', label="Anomaly" if name == 1 else "Normal")

        axes.hlines(threshold, axes.get_xlim()[0], axes.get_xlim()[1], colors="b", zorder=100, label='Threshold')
        axes.legend()

        plt.title("Anomalies")
        plt.ylabel("Error")
        plt.xlabel("Data")
        plt.show()

    def draw_error(self, error, threshold):
        plt.plot(error, marker='o', ms=3.5, linestyle='',
                 label='Point')

        plt.hlines(threshold, xmin=0, xmax=len(error)-1, colors="b", zorder=100, label='Threshold')
        plt.legend()
        plt.title("Reconstruction error")
        plt.ylabel("Error")
        plt.xlabel("Data")
        plt.show()

```

Figure 4-5. Visualization helpers

You will use the example of credit card data to detect whether a transaction is normal/expected or abnormal/anomaly. Figure 4-6 shows the data being loaded into a Pandas dataframe.

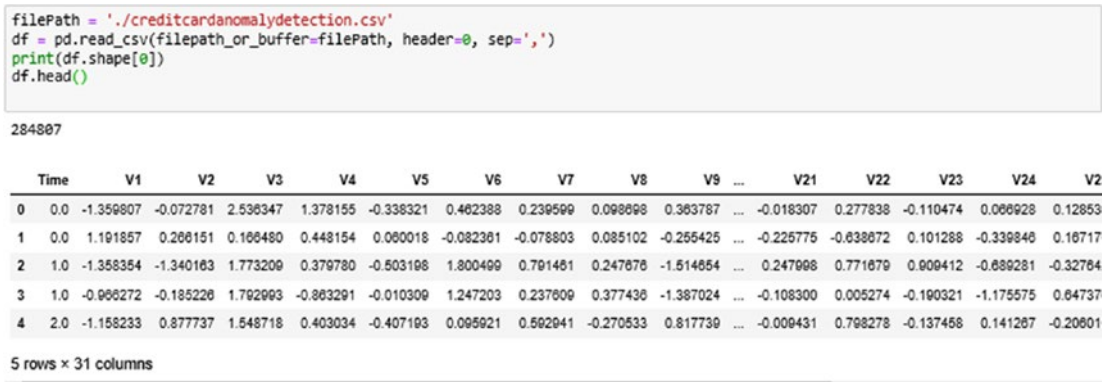


Figure 4-6. Examining the Pandas dataframe

You will collect 20k normal and 400 abnormal records. You can pick different ratios to try, but in general more normal data examples are better because you want to teach your autoencoder what normal data looks like. Too much abnormal data in training will train the autoencoder to learn that the anomalies are actually normal, which goes against your goal. Figure 4-7 shows sampling the dataframe and choosing the majority of normal data.

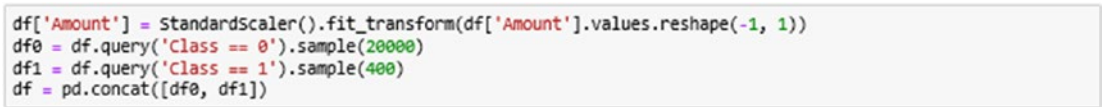


Figure 4-7. Sampling the dataframe and choosing the majority of normal data

You split the dataframe into training and testing data sets (80-20 split). Figure 4-8 shows the code to split the data into the train and test subsets.



Figure 4-8. Splitting the data into test and train sets, using 20% as holdout test data

Now it's time to create a simple neural network model with just an encoder and decoder layer. You will encode the 29 columns of the input credit card dataset into 12 features using the encoder. The decoder expands the 12 back into the 29 features. Figure 4-9 shows the code to create the neural network.

```
encoding_dim = 12
input_dim = x_train.shape[1]

inputArray = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='relu')(inputArray)
decoded = Dense(input_dim, activation='softmax')(encoded)

autoencoder = Model(inputArray, decoded)
autoencoder.summary()
```

WARNING:tensorflow:From C:\ProgramData\Anaconda3\lib\site-packages\tensorflow\python\framework\op_def_library.py:263: colocate_with (from tensorflow.python.framework.ops) is deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 29)	0
dense_1 (Dense)	(None, 12)	360
dense_2 (Dense)	(None, 29)	377

Total params: 737
 Trainable params: 737
 Non-trainable params: 0

Figure 4-9. *Creating the simple autoencoder neural network*

If you look at the code in Figure 4-9, you will see two different activation functions, namely relu and softmax. So what are they?

RELU, the Rectified Linear Unit, is the most commonly used activation function in deep learning models. The function returns 0 if it receives any negative input, but for any positive value xx it returns that value back. So it can be written as

$$f(x)=\max(0,x).$$

Softmax, the Softmax function, outputs a vector that represents the probability distributions of a list of potential outcomes. The probabilities always add up to 1.

Needless to say, there are several activation functions available and you can refer to the Keras documentation to look at the options at <https://keras.io/activations/>.

Now, compile the model using RMSprop as the optimizer and mean squared error for the loss computation. The RMSprop optimizer is similar to the gradient descent algorithm with momentum. A metric function is similar to a loss function, except that the results from evaluating a metric are not used when training the model. You may use

any of the loss functions as a metric function, as listed in <https://keras.io/losses/>. Figure 4-10 shows the code to compile the model using mean absolute error and accuracy as metrics.

```
autoencoder.compile(optimizer=RMSprop(),  
                    loss='mean_squared_error',  
                    metrics=['mae', 'accuracy'])
```

Figure 4-10. *Compiling the model*

Now you can start training the model using the training dataset to validate the model at every step. Choose 32 as the batchsize and 20 epochs. Figure 4-11 shows the code to train the model, which is the most time consuming part of the process.

```
batch_size = 32  
epochs = 20  
  
history = autoencoder.fit(x_train, x_train,  
                        batch_size=batch_size,  
                        epochs=epochs,  
                        verbose=1,  
                        shuffle=True,  
                        validation_data=(x_test, x_test),  
                        callbacks=[TensorBoard(log_dir='../logs/autoencoder1')])
```

Figure 4-11. *Training the model*

As you see, the training process outputs the loss and accuracy, as well as the validation loss and validation accuracy at each epoch. Figure 4-12 shows the output of the training step.

```

Epoch 3/20
16320/16320 [=====] - 2s 106us/step - loss: 1.4586 - mean_absolute_error: 0.6595 - acc: 0.6291 - val_1
oss: 1.6319 - val_mean_absolute_error: 0.6643 - val_acc: 0.6525
Epoch 4/20
16320/16320 [=====] - 2s 106us/step - loss: 1.4536 - mean_absolute_error: 0.6582 - acc: 0.6710 - val_1
oss: 1.6290 - val_mean_absolute_error: 0.6636 - val_acc: 0.6848
Epoch 5/20
16320/16320 [=====] - 2s 107us/step - loss: 1.4514 - mean_absolute_error: 0.6578 - acc: 0.6953 - val_1
oss: 1.6275 - val_mean_absolute_error: 0.6633 - val_acc: 0.7071
Epoch 6/20
16320/16320 [=====] - 2s 108us/step - loss: 1.4502 - mean_absolute_error: 0.6575 - acc: 0.7140 - val_1
oss: 1.6266 - val_mean_absolute_error: 0.6631 - val_acc: 0.7206
Epoch 7/20
16320/16320 [=====] - 2s 106us/step - loss: 1.4493 - mean_absolute_error: 0.6574 - acc: 0.7300 - val_1
oss: 1.6258 - val_mean_absolute_error: 0.6630 - val_acc: 0.7373
Epoch 8/20
16320/16320 [=====] - 2s 110us/step - loss: 1.4486 - mean_absolute_error: 0.6573 - acc: 0.7474 - val_1
oss: 1.6253 - val_mean_absolute_error: 0.6630 - val_acc: 0.7488
Epoch 9/20
16320/16320 [=====] - 2s 112us/step - loss: 1.4482 - mean_absolute_error: 0.6572 - acc: 0.7580 - val_1
oss: 1.6249 - val_mean_absolute_error: 0.6629 - val_acc: 0.7593
Epoch 10/20
16320/16320 [=====] - 2s 115us/step - loss: 1.4478 - mean_absolute_error: 0.6572 - acc: 0.7670 - val_1
oss: 1.6246 - val_mean_absolute_error: 0.6629 - val_acc: 0.7689
Epoch 11/20
16320/16320 [=====] - 2s 113us/step - loss: 1.4476 - mean_absolute_error: 0.6572 - acc: 0.7722 - val_1
oss: 1.6244 - val_mean_absolute_error: 0.6628 - val_acc: 0.7691
Epoch 12/20
16320/16320 [=====] - 2s 114us/step - loss: 1.4473 - mean_absolute_error: 0.6571 - acc: 0.7769 - val_1
oss: 1.6242 - val_mean_absolute_error: 0.6628 - val_acc: 0.7723
Epoch 13/20
16320/16320 [=====] - 2s 109us/step - loss: 1.4472 - mean_absolute_error: 0.6571 - acc: 0.7820 - val_1
oss: 1.6241 - val_mean_absolute_error: 0.6628 - val_acc: 0.7748
Epoch 14/20
16320/16320 [=====] - 2s 110us/step - loss: 1.4470 - mean_absolute_error: 0.6571 - acc: 0.7847 - val_1
oss: 1.6239 - val_mean_absolute_error: 0.6628 - val_acc: 0.7775
Epoch 15/20
16320/16320 [=====] - 2s 117us/step - loss: 1.4469 - mean_absolute_error: 0.6571 - acc: 0.7871 - val_1
oss: 1.6238 - val_mean_absolute_error: 0.6628 - val_acc: 0.7789
Epoch 16/20
16320/16320 [=====] - 2s 103us/step - loss: 1.4468 - mean_absolute_error: 0.6571 - acc: 0.7881 - val_1
oss: 1.6237 - val_mean_absolute_error: 0.6628 - val_acc: 0.7792
Epoch 17/20
16320/16320 [=====] - 2s 101us/step - loss: 1.4468 - mean_absolute_error: 0.6571 - acc: 0.7897 - val_1
oss: 1.6237 - val_mean_absolute_error: 0.6628 - val_acc: 0.7850
Epoch 18/20
16320/16320 [=====] - 2s 100us/step - loss: 1.4467 - mean_absolute_error: 0.6570 - acc: 0.7908 - val_1
oss: 1.6236 - val_mean_absolute_error: 0.6627 - val_acc: 0.7828
Epoch 19/20
16320/16320 [=====] - 2s 103us/step - loss: 1.4467 - mean_absolute_error: 0.6570 - acc: 0.7950 - val_1
oss: 1.6235 - val_mean_absolute_error: 0.6627 - val_acc: 0.7826
Epoch 20/20
16320/16320 [=====] - 2s 107us/step - loss: 1.4466 - mean_absolute_error: 0.6570 - acc: 0.7955 - val_1
oss: 1.6235 - val_mean_absolute_error: 0.6627 - val_acc: 0.7853

```

Figure 4-12. Showing the progress of the training phase

Figure 4-13 is a graph of the model as shown by TensorBoard.

Main Graph

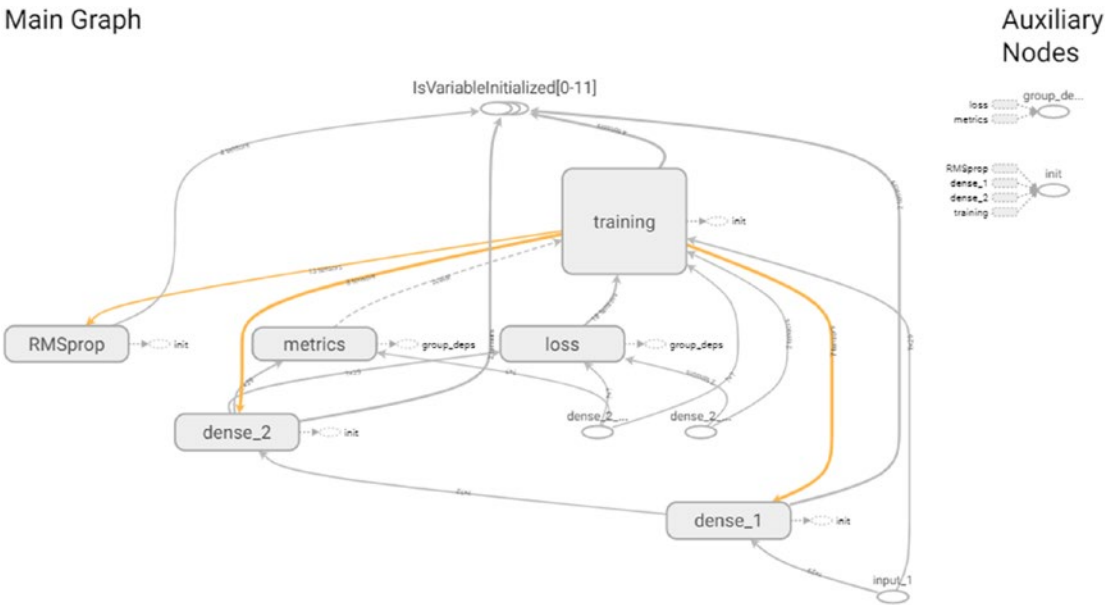


Figure 4-13. Model graph shown in TensorBoard

Figure 4-14 shows the plotting of the accuracy during the training process through the epochs of training.

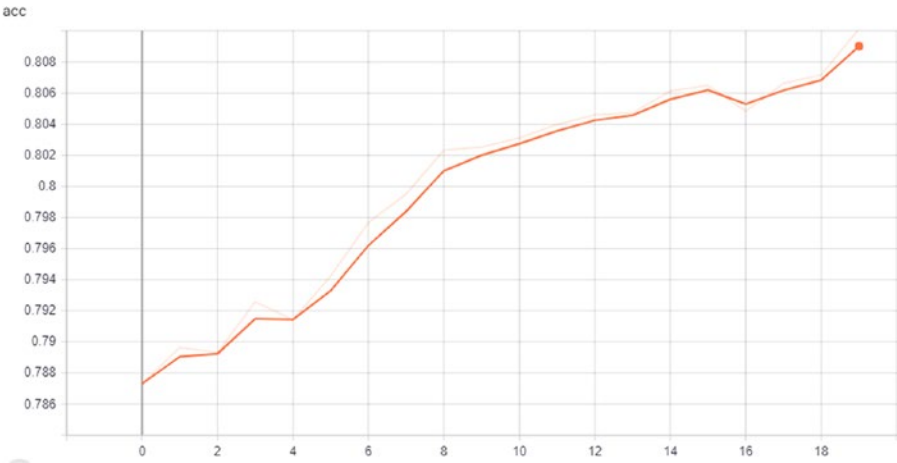


Figure 4-14. Plotting of accuracy shown in TensorBoard

Figure 4-17 shows the plotting of the accuracy of validation during the training process through the epochs of training.

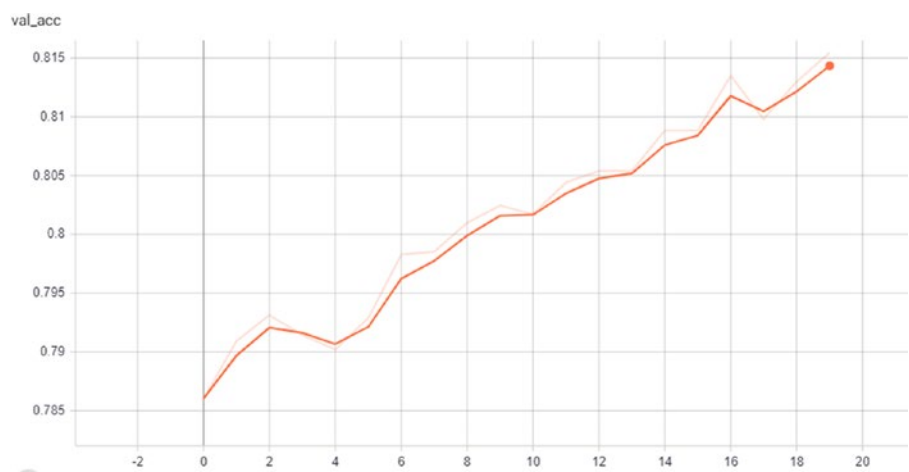


Figure 4-17. *Plotting of validation accuracy shown in TensorBoard*

Figure 4-18 shows the plotting of the loss of validation during the training process through the epochs of training.

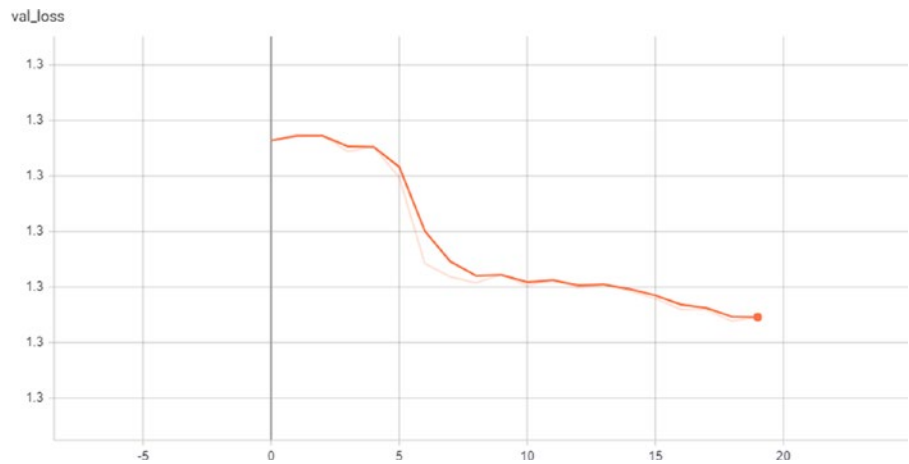


Figure 4-18. *Plotting of validation loss shown in TensorBoard*

Now that the training process is complete, let’s evaluate the model for loss and accuracy. Figure 4-19 shows that the accuracy is 0.81, which is pretty good. It also shows the code to evaluate the model.


```

score = autoencoder.evaluate(x_test, x_test, verbose=1)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

4080/4080 [=====] - 0s 56us/step
Test loss: 1.3027283556321088
Test accuracy: 0.8154411764705882

```

Figure 4-19. Code to evaluate the model

The next step is to calculate the errors, and detect and also plot the anomalies and errors. Choose a threshold of 10. Figure 4-20 shows the code to measure anomalies based on that threshold.

```

threshold=10.00
y_pred = autoencoder.predict(x_test)
y_dist = np.linalg.norm(x_test - y_pred, axis=-1)
z = zip(y_dist >= threshold, y_dist)
y_label=[]
error = []
for idx, (is_anomaly, y_dist) in enumerate(z):
    if is_anomaly:
        y_label.append(1)
    else:
        y_label.append(0)
    error.append(y_dist)

```

Figure 4-20. Code to measure anomalies based on a threshold

Let's delve deeper into the code shown above because this will be seen throughout the chapter when you classify data points as anomalies or normal. As you can see, this is based on a special parameter called the threshold. You are simply looking at the error (difference between actual and predicted) and comparing it to the threshold. First, calculate the precision and recall for threshold = 10. Figure 4-21a shows the code to show the precision and recall.

```

print(classification_report(y_test,y_label))

```

	precision	recall	f1-score	support
0	1.00	0.97	0.98	3987
1	0.41	0.86	0.56	93
accuracy			0.97	4080
macro avg	0.71	0.92	0.77	4080
weighted avg	0.98	0.97	0.97	4080

Figure 4-21a. Code to show the precision and recall

Let’s also calculate for thresholds = 1, 5, 15. See Figures 4-21b, 4-21c, and 4-21d.
Threshold = 1.0

```
print(classification_report(y_test,y_label))
```

	precision	recall	f1-score	support
0	0.00	0.00	0.00	3987
1	0.02	1.00	0.04	93
accuracy			0.02	4080
macro avg	0.01	0.50	0.02	4080
weighted avg	0.00	0.02	0.00	4080

Figure 4-21b. Code to show the precision and recall for threshold = 1.0

Threshold = 5.0

```
print(classification_report(y_test,y_label))
```

	precision	recall	f1-score	support
0	1.00	0.75	0.86	3987
1	0.08	0.97	0.15	93
accuracy			0.76	4080
macro avg	0.54	0.86	0.51	4080
weighted avg	0.98	0.76	0.84	4080

Figure 4-21c. Code to show the precision and recall for threshold = 5.0

Threshold = 15.0

```
print(classification_report(y_test,y_label))
```

	precision	recall	f1-score	support
0	0.99	0.99	0.99	3987
1	0.57	0.66	0.61	93
accuracy			0.98	4080
macro avg	0.78	0.82	0.80	4080
weighted avg	0.98	0.98	0.98	4080

Figure 4-21d. Code to show the precision and recall for threshold = 15.0

If you observe the four classification reports, you can see that the precision and recall columns are not good (note the very low values for precision and recall in row 0 and row 1) for threshold = 1 or 5. They look better for threshold = 10 or 15. In fact, threshold = 10 looks pretty good with a good recall and also higher precision than for threshold = 1 or 5.

Picking a threshold is a matter of experimentation in this and other models and changes as per the data being trained on.

Compute the AUC (Area Under the Curve, 0.0 to 1.0) which comes up as 0.86. Figure 4-21e shows the code to show AUC.

```
roc_auc_score(y_test, y_label)
0.8650574043059298
```

Figure 4-21e. Code to show AUC

You can now visualize the confusion matrix to see how well you did with the model. Figure 4-22 shows the confusion matrix.

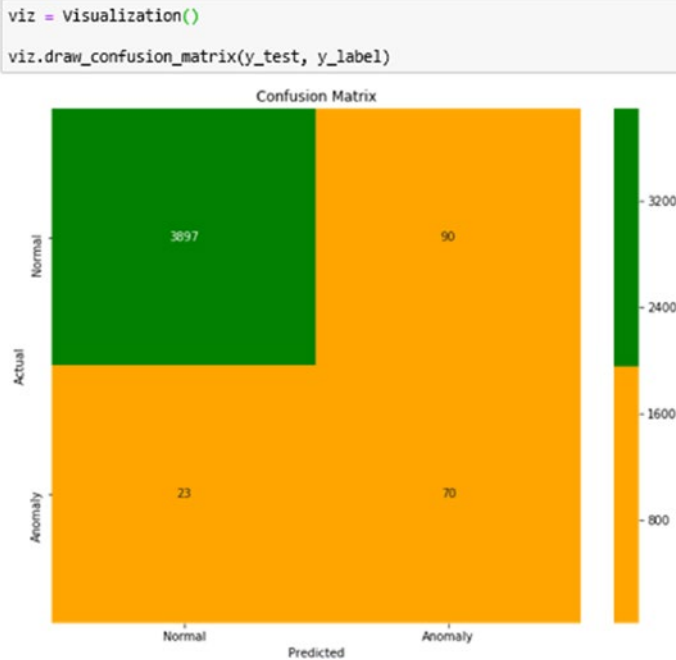


Figure 4-22. Confusion matrix

Now, using the predictions of the labels (normal or anomaly), you can plot the anomalies in comparison to the normal data points. Figure 4-23 shows the anomalies based on the threshold.

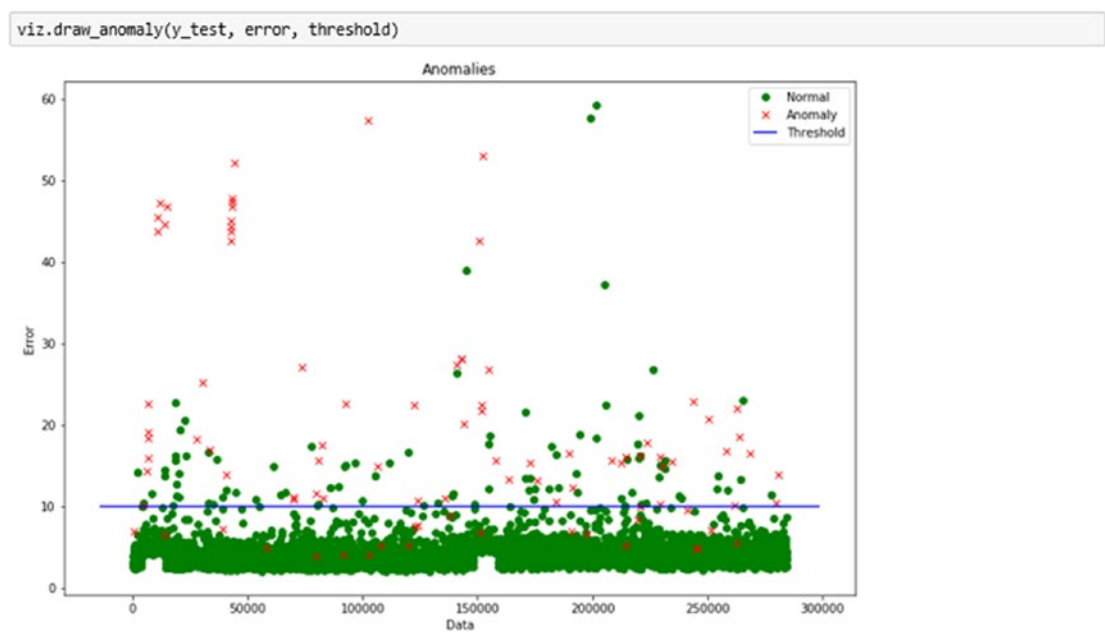


Figure 4-23. *Anomalies based on the threshold*

Sparse Autoencoders

In the above example of a simple autoencoder, the representations were only constrained by the size of the hidden layer (12). In such a situation, what typically happens is that the hidden layer is learning an approximation of PCA (principal component analysis). But another way to constrain the representations to be compact is to add a sparsity constraint on the activity of the hidden representations, so fewer units would fire at a given time. In Keras, this can be done by adding an `activity_regularizer` to your dense layer.

The difference between the simple and sparse autoencoders is mostly due to the regularization term being added to the loss during training.

```
from keras import regularizers
```

You will use the same credit card dataset as in the simple autoencoder example above. You will use the credit card data to detect whether a transaction is normal/expected or abnormal/anomaly. Shown below is the data being loaded into a Pandas dataframe.

Then, you will collect 20k normal and 400 abnormal records. You can pick different ratios to try, but in general more normal data examples are better because you want to teach your autoencoder what normal data looks like. Too much abnormal data in training will train the autoencoder to learn that the anomalies are actually normal, which goes against your goal. Split the dataframe into training and testing data sets (80-20 split).

Now it's time to create a neural network model with just an encoder and decoder layer. You will encode the 29 columns of the input credit card dataset into 12 features using the encoder. The decoder will expand the 12 back into 29 features. The key difference compared to the simple autoencoder is the activity regularizer to accommodate the sparse autoencoder. Figure 4-24 shows the code to create the neural network.

```
encoding_dim = 12
input_dim = x_train.shape[1]

inputArray = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='relu',
                activity_regularizer=regularizers.l1(10e-5))(inputArray)

decoded = Dense(input_dim, activation='softmax')(encoded)

autoencoder = Model(inputArray, decoded)
autoencoder.summary()
```

WARNING:tensorflow:From C:\ProgramData\Anaconda3\lib\site-packages\tensorflow\python\framework\op_def_library.py:263: colocate_ with (from tensorflow.python.framework.ops) is deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 29)	0
dense_1 (Dense)	(None, 12)	360
dense_2 (Dense)	(None, 29)	377

Total params: 737
Trainable params: 737
Non-trainable params: 0

Figure 4-24. Code to create the neural network

Figure 4-25 shows the graph of the model as visualized by TensorBoard.

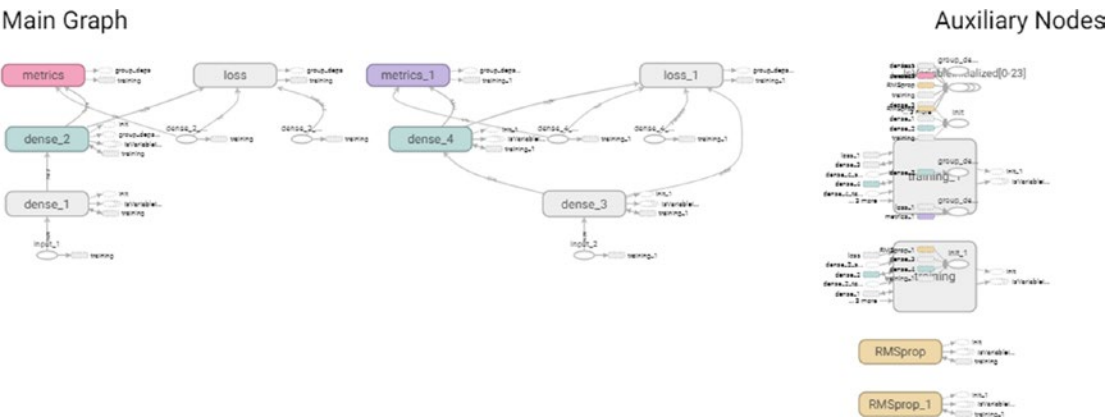


Figure 4-25. Model graph created by TensorBoard

Deep Autoencoders

You do not have to limit yourself to a single layer as encoder or decoder; you can use a stack of layers. It's not a good idea to use too many hidden layers, and how many layers depends on the use case, so you have to play with it to seek the optimal number of layers and the compressions.

The only thing that really changes is the number of layers. Shown below is the simple autoencoder with multiple layers.

You will use the example of credit card data to detect whether a transaction is normal/expected or abnormal/anomaly. Shown below is the data being loaded into Pandas dataframe.

You will collect 20k normal and 400 abnormal records. You can pick different ratios to try, but in general more normal data examples are better because you want to teach your autoencoder what normal data looks like. Too much abnormal data in training will train the autoencoder to learn that the anomalies are actually normal, which goes against your goal. Split the dataframe into training and testing data sets (80-20 split).

Now it's time to create a deep neural network model with three layers for the encoder layer and three layers as part of decoder layer. You will encode the 29 columns of the input credit card dataset into 16, then 8, and then 4 features using the encoder. The decoder expands the 4 back into the 8 and then 16 and then finally into 29 features. Figure 4-26 shows the code to create the neural network.

```
#deep autoencoder
logfilename = "deepautoencoder"

encoding_dim = 16
input_dim = x_train.shape[1]

inputArray = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='relu')(inputArray)
encoded = Dense(8, activation='relu')(encoded)
encoded = Dense(4, activation='relu')(encoded)

decoded = Dense(8, activation='relu')(encoded)
decoded = Dense(encoding_dim, activation='relu')(decoded)
decoded = Dense(input_dim, activation='softmax')(decoded)

autoencoder = Model(inputArray, decoded)
autoencoder.summary()
```

Layer (type)	Output Shape	Param #
input_7 (InputLayer)	(None, 29)	0
dense_15 (Dense)	(None, 16)	480
dense_16 (Dense)	(None, 8)	136
dense_17 (Dense)	(None, 4)	36
dense_18 (Dense)	(None, 8)	40
dense_19 (Dense)	(None, 16)	144
dense_20 (Dense)	(None, 29)	493
Total params: 1,329		
Trainable params: 1,329		
Non-trainable params: 0		

Figure 4-26. Code to create the neural network

Figure 4-27 shows the graph of the model as visualized by TensorBoard.

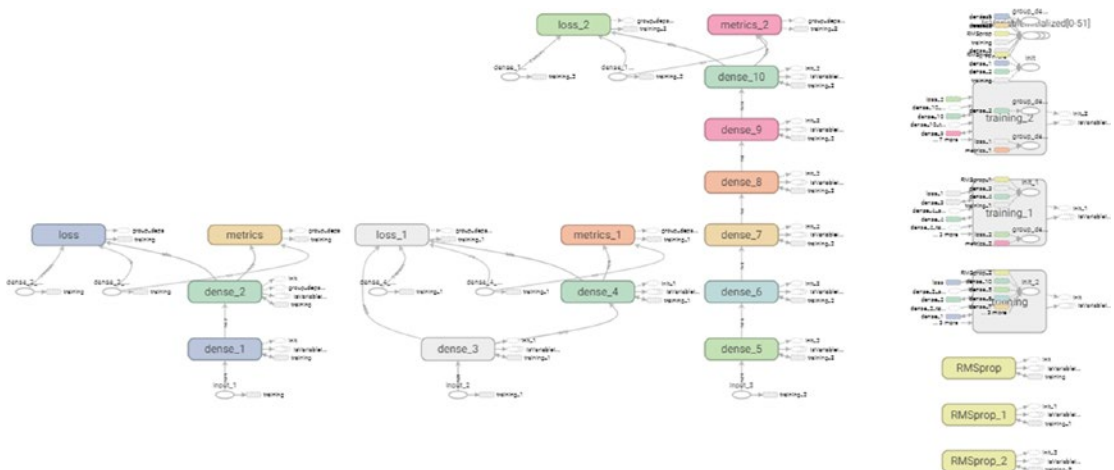


Figure 4-27. Model graph shown in TensorBoard

Convolutional Autoencoders

Whenever your inputs are images, it makes sense to use convolutional neural networks (convnets or CNNs) as encoders and decoders. In practical settings, autoencoders applied to images are always convolutional autoencoders because they simply perform much better.

Let's implement one. The encoder will consist in a stack of Conv2D and MaxPooling2D layers (max pooling is being used for spatial down-sampling), while the decoder will consist in a stack of Conv2D and UpSampling2D layers.

Figure 4-28 shows the basic code to import all necessary packages in a Jupyter notebook. Also note the versions of the various packages.

```
import keras
from keras import optimizers
from keras import losses
from keras.models import Sequential, Model
from keras.layers import Dense, Input, Dropout, Embedding, LSTM
from keras.optimizers import RMSprop, Adam, Nadam
from keras.preprocessing import sequence
from keras.callbacks import TensorBoard
from keras import regularizers

import sklearn
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, roc_auc_score
from sklearn.preprocessing import MinMaxScaler

import seaborn as sns
import pandas as pd
import numpy as np
import matplotlib

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
%matplotlib inline

import tensorflow
import sys
print("Python: ", sys.version)

print("pandas: ", pd.__version__)
print("numpy: ", np.__version__)
print("seaborn: ", sns.__version__)
print("matplotlib: ", matplotlib.__version__)
print("sklearn: ", sklearn.__version__)
print("Keras: ", keras.__version__)
print("Tensorflow: ", tensorflow.__version__)

Using TensorFlow backend.

Python: 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)]
pandas: 0.24.2
numpy: 1.16.3
seaborn: 0.9.0
matplotlib: 3.0.3
sklearn: 0.20.3
Keras: 2.2.4
Tensorflow: 1.13.1
```

Figure 4-28. Importing packages in a Jupyter notebook

You will use the mnist images data set for this purpose. Mnist contains images for the digits 0 to 9 and is used for many different use cases. Figure 4-29 shows the code to load MNIST data.

```
from keras.datasets import mnist
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()
```

Figure 4-29. Code to load MNIST data

Split the dataset into training and testing subsets. You must also reshape the data to 28X28 images. Figure 4-30 shows the code to transform the images from MNIST.

```
from keras.datasets import mnist
import numpy as np

(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1)) # adapt this if using 'channels_first' image data format
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1)) # adapt this if using 'channels_first' image data format
```

Figure 4-30. Code to transform the images from MNIST

Create a CNN model with Convolutions and MaxPool layers. Figure 4-31 shows the code to create the neural network.

```
from keras.layers import Input, Dense, Conv2D, MaxPooling2D, UpSampling2D
from keras.models import Model
from keras import backend as K

#cnv autoencoder
logfile = "cnvautoencoder2"

input_img = Input(shape=(28, 28, 1)) # adapt this if using `channels_first` image data format

x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (4, 4, 8) i.e. 128-dimensional

x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)

autoencoder.summary()
```

WARNING:tensorflow:From C:\ProgramData\Anaconda3\lib\site-packages\tensorflow\python\framework\op_def_library.py:263: colocate_with (from tensorflow.python.framework.ops) is deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 28, 28, 1)	0
conv2d_1 (Conv2D)	(None, 28, 28, 16)	160
max_pooling2d_1 (MaxPooling2D)	(None, 14, 14, 16)	0
conv2d_2 (Conv2D)	(None, 14, 14, 8)	1160
max_pooling2d_2 (MaxPooling2D)	(None, 7, 7, 8)	0
conv2d_3 (Conv2D)	(None, 7, 7, 8)	584
max_pooling2d_3 (MaxPooling2D)	(None, 4, 4, 8)	0
conv2d_4 (Conv2D)	(None, 4, 4, 8)	584
up_sampling2d_1 (UpSampling2D)	(None, 8, 8, 8)	0
conv2d_5 (Conv2D)	(None, 8, 8, 8)	584
up_sampling2d_2 (UpSampling2D)	(None, 16, 16, 8)	0
conv2d_6 (Conv2D)	(None, 14, 14, 16)	1168
up_sampling2d_3 (UpSampling2D)	(None, 28, 28, 16)	0
conv2d_7 (Conv2D)	(None, 28, 28, 1)	145

Total params: 4,385
Trainable params: 4,385
Non-trainable params: 0

Figure 4-31. Code to create the neural network

Compile the model using RMSprop as the optimizer and mean squared error for the loss computation. The RMSprop optimizer is similar to the gradient descent algorithm with momentum. Figure 4-32 shows the code to compile the model.

```
autoencoder.compile(optimizer=RMSprop(),
                    loss='mean_squared_error',
                    metrics=['mae', 'accuracy'])
```

Figure 4-32. Code to compile the model

Now you can start training the model using the training dataset while using the validation dataset to validate the model at every step. Choose 32 as the batchsize and 20 epochs. The training process outputs the loss and accuracy as well as the validation loss and validation accuracy at each epoch. Figure 4-33 shows the model being trained.

```
batch_size = 32
epochs = 20

history = autoencoder.fit(x_train, x_train,
                        batch_size=batch_size,
                        epochs=epochs,
                        verbose=1,
                        shuffle=True,
                        validation_data=(x_test, x_test),
                        callbacks=[TensorBoard(log_dir='../logs/{0}'.format(logfilename))])
```

```
20
Epoch 15/20
60000/60000 [=====] - 12s 194us/step - loss: 0.0116 - acc: 0.8129 - val_loss: 0.0105 - val_acc: 0.81
24
Epoch 16/20
60000/60000 [=====] - 12s 196us/step - loss: 0.0114 - acc: 0.8130 - val_loss: 0.0117 - val_acc: 0.81
09
Epoch 17/20
60000/60000 [=====] - 11s 183us/step - loss: 0.0113 - acc: 0.8131 - val_loss: 0.0108 - val_acc: 0.81
30
Epoch 18/20
60000/60000 [=====] - 11s 188us/step - loss: 0.0112 - acc: 0.8131 - val_loss: 0.0103 - val_acc: 0.81
27
Epoch 19/20
60000/60000 [=====] - 11s 190us/step - loss: 0.0110 - acc: 0.8132 - val_loss: 0.0107 - val_acc: 0.81
28
Epoch 20/20
60000/60000 [=====] - 12s 192us/step - loss: 0.0109 - acc: 0.8132 - val_loss: 0.0103 - val_acc: 0.81
26
```

Figure 4-33. The model being trained

Now that the training process is complete, let's evaluate the model for loss and accuracy. Figure 4-34 shows that the accuracy is 0.81, which is pretty good. It also shows the code to evaluate the model.

```
score = autoencoder.evaluate(x_test, x_test, verbose=1)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
10000/10000 [=====] - 1s 68us/step
Test loss: 0.010284392775595189
Test accuracy: 0.8126302285194397
```

Figure 4-34. Code to evaluate the model

The next step is to use the model to generate the output images for the testing subset. This will show how well the reconstruction phase is going. Figure 4-35 shows the code to predict based on the model.

```
decoded_imgs = autoencoder.predict(x_test)

n = 10
plt.figure(figsize=(20, 4))
for i in range(1, n):
    # display original
    ax = plt.subplot(2, n, i)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



Figure 4-35. Code to predict based on the model

You can also see how the encoder phase is working by displaying the test subset images in this phase. Figure 4-36 shows the code to display the encoded images.

```

encoder = Model(input_img, encoded)
encoded_imgs = encoder.predict(x_test)
n = 10
plt.figure(figsize=(20, 8))
for i in range(1, n):
    ax = plt.subplot(1, n, i)
    plt.imshow(encoded_imgs[i].reshape(4, 4 * 8).T)
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()

```

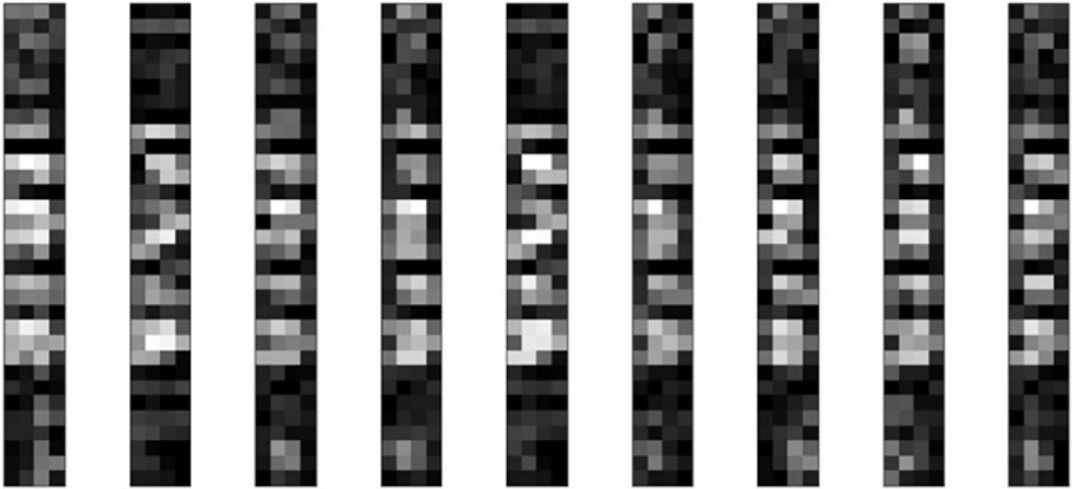


Figure 4-36. Code to display encoded images

Figure 4-37 shows the graph of the model as visualized by TensorBoard.

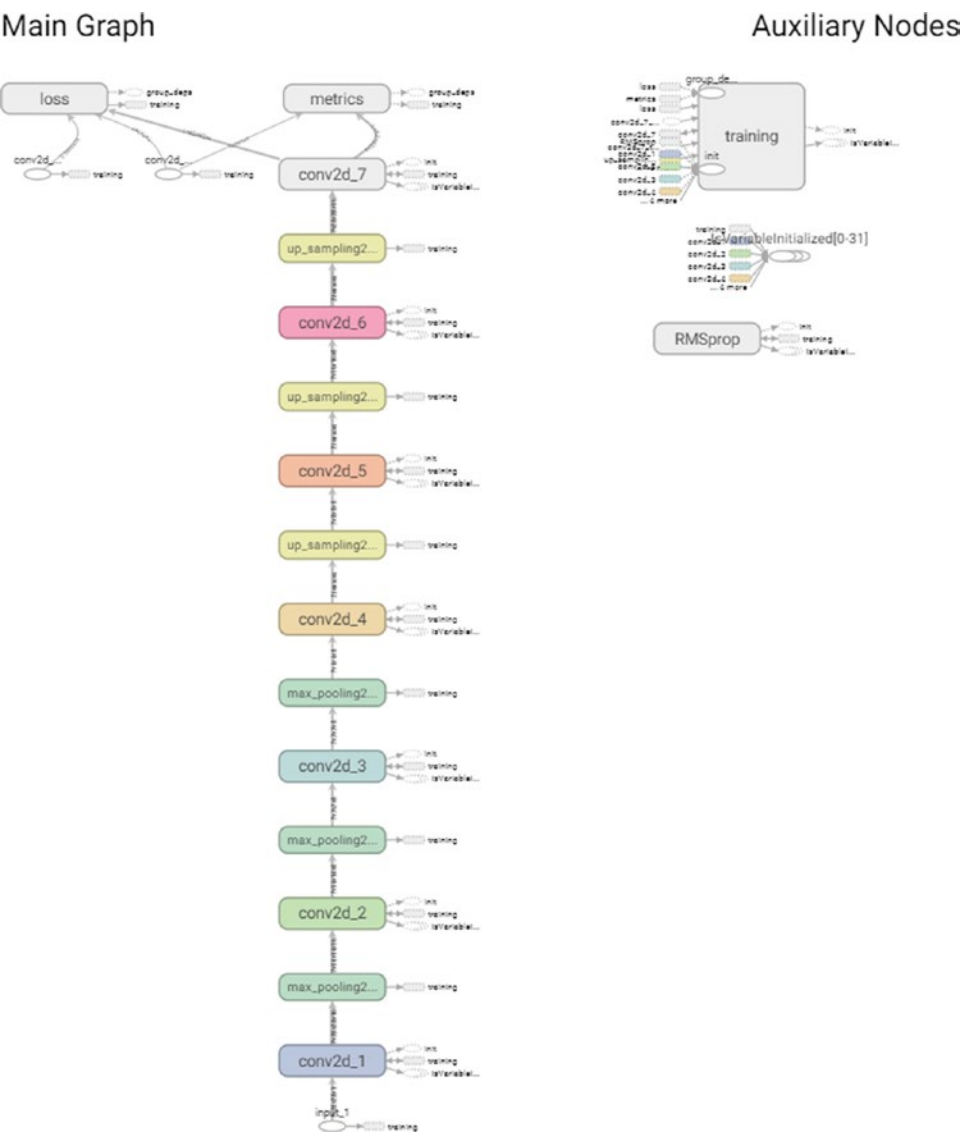


Figure 4-37. A model graph shown in TensorBoard

Figure 4-38 shows the plotting of the accuracy during the training process through the epochs of training.

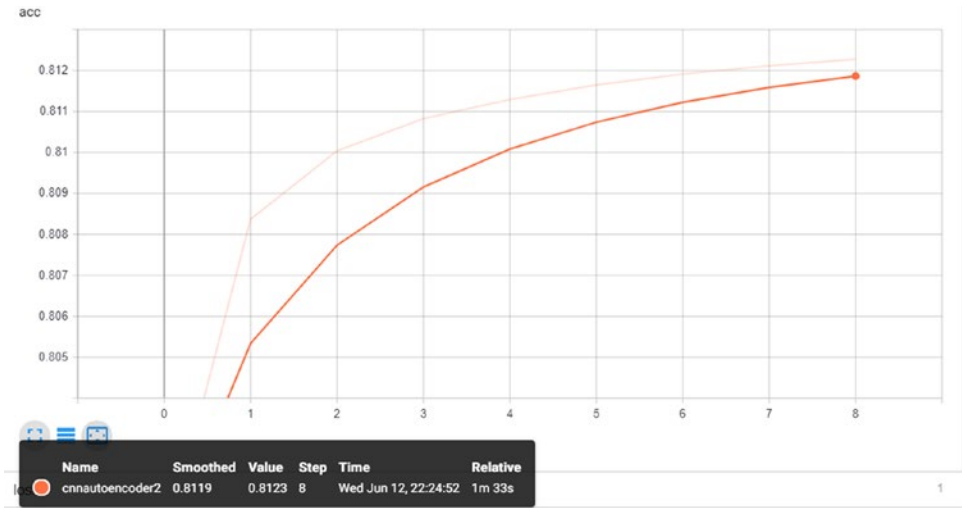


Figure 4-38. Plotting of accuracy shown in TensorBoard

Figure 4-39 shows the plotting of the loss during the training process through the epochs of training.

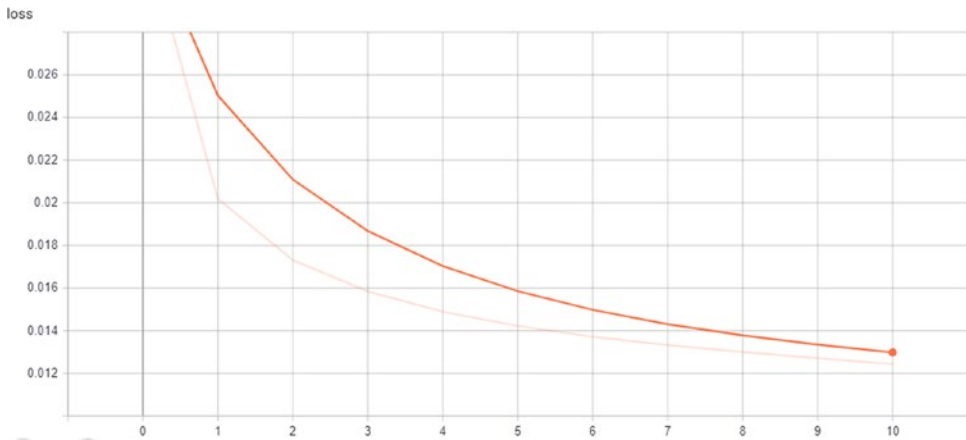


Figure 4-39. Plotting of loss shown in TensorBoard

Figure 4-40 shows the plotting of the accuracy of validation during the training process through the epochs of training.

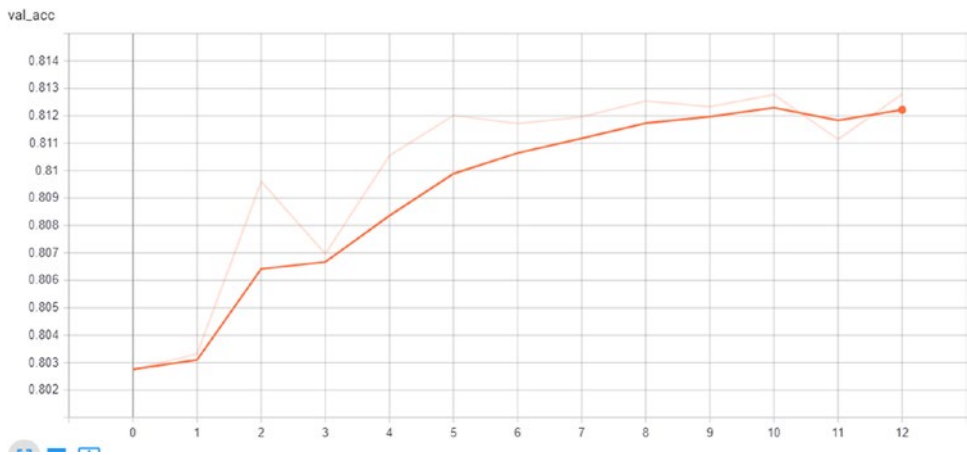


Figure 4-40. Plotting of validation accuracy shown in TensorBoard

Figure 4-41 shows the plotting of the loss of validation during the training process through the epochs of training.



Figure 4-41. Plotting of validation loss shown in TensorBoard

Denoising Autoencoders

You can force the autoencoder to learn useful features by adding random noise to its inputs and making it recover the original noise-free data. This way the autoencoder can't simply copy the input to its output because the input also contains random noise. The autoencoder will remove noise and produce the underlying meaningful data. This is called a denoising autoencoder. Figure 4-42 shows a depiction of a denoising autoencoder.

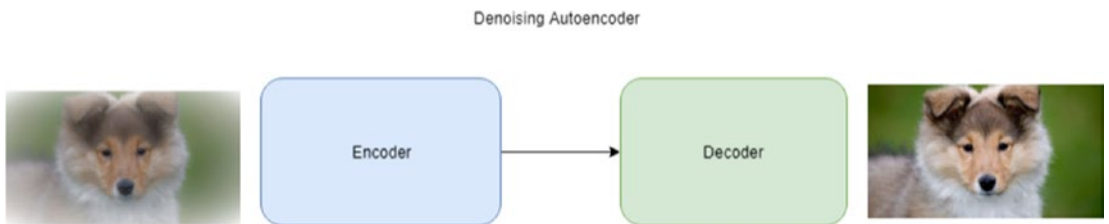


Figure 4-42. *Depiction of a denoising autoencoder*

Other example is a security monitoring camera capturing some kind of hazy unclear picture, maybe in the dark or during adverse weather, causing a noisy image.

The logic behind the denoising autoencoder is that if you have trained your encoder on good normal images, and the noise, when it comes as part of the input, is not really a salient characteristic, it is possible to detect and remove such noise.

Figure 4-43 shows the basic code to import all necessary packages. Also note the versions of the various packages.

```

import keras
from keras import optimizers
from keras import losses
from keras.models import Sequential, Model
from keras.layers import Dense, Input, Dropout, Embedding, LSTM
from keras.optimizers import RMSprop, Adam, Nadam
from keras.preprocessing import sequence
from keras.callbacks import TensorBoard
from keras import regularizers

import sklearn
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, roc_auc_score
from sklearn.preprocessing import MinMaxScaler

import seaborn as sns
import pandas as pd
import numpy as np
import matplotlib

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
%matplotlib inline

import tensorflow
import sys
print("Python: ", sys.version)

print("pandas: ", pd.__version__)
print("numpy: ", np.__version__)
print("seaborn: ", sns.__version__)
print("matplotlib: ", matplotlib.__version__)
print("sklearn: ", sklearn.__version__)
print("Keras: ", keras.__version__)
print("Tensorflow: ", tensorflow.__version__)

```

Using TensorFlow backend.

```

Python: 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)]
pandas: 0.24.2
numpy: 1.16.3
seaborn: 0.9.0
matplotlib: 3.0.3
sklearn: 0.20.3
Keras: 2.2.4
Tensorflow: 1.13.1

```

Figure 4-43. Code to import packages

You will use the mnist images data set for this purpose. Mnist contains images for the digits 0 to 9 and is used for many different use cases. Figure 4-44 shows the code to load MNIST images.

```

from keras.datasets import mnist
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()

```

Figure 4-44. Code to load MNIST images

Split the dataset into training and testing subsets. Also, reshape the data to 28X28 images. Figure 4-45 shows the code to load and reshape images.

```
from keras.datasets import mnist
import numpy as np

(x_train, _), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1)) # adapt this if using `channels_first` image data format
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1)) # adapt this if using `channels_first` image data format

noise_factor = 0.3
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)

x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)

print(x_train_noisy.shape)
print(x_test_noisy.shape)
print(y_test.shape)

(60000, 28, 28, 1)
(10000, 28, 28, 1)
(10000,)
```

Figure 4-45. Code to load and reshape images

Figure 4-46 shows the code to display the images.

```
n = 11
plt.figure(figsize=(20, 2))
for i in range(1, n):
    ax = plt.subplot(1, n, i)
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



Figure 4-46. Code to display the images

Create a CNN model with Convolutions and MaxPool layers. Figure 4-47 shows the code to create the neural network.

```
from keras.layers import Input, Dense, Conv2D, MaxPooling2D, UpSampling2D
from keras.models import Model
from keras import backend as K

# cnn autoencoder
logfile = "DenoisingAutoencoder2"

input_img = Input(shape=(28, 28, 1)) # adapt this if using `channels_first` image data format

x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (4, 4, 8) i.e. 128-dimensional

x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)

autoencoder.summary()
```

Layer (type)	Output Shape	Param #
=====		
input_2 (InputLayer)	(None, 28, 28, 1)	0
conv2d_8 (Conv2D)	(None, 28, 28, 16)	160
max_pooling2d_4 (MaxPooling2	(None, 14, 14, 16)	0
conv2d_9 (Conv2D)	(None, 14, 14, 8)	1160
max_pooling2d_5 (MaxPooling2	(None, 7, 7, 8)	0
conv2d_10 (Conv2D)	(None, 7, 7, 8)	584
max_pooling2d_6 (MaxPooling2	(None, 4, 4, 8)	0
conv2d_11 (Conv2D)	(None, 4, 4, 8)	584
up_sampling2d_4 (UpSampling2	(None, 8, 8, 8)	0
conv2d_12 (Conv2D)	(None, 8, 8, 8)	584
up_sampling2d_5 (UpSampling2	(None, 16, 16, 8)	0
conv2d_13 (Conv2D)	(None, 14, 14, 16)	1168
up_sampling2d_6 (UpSampling2	(None, 28, 28, 16)	0
conv2d_14 (Conv2D)	(None, 28, 28, 1)	145
=====		
Total params: 4,385		
Trainable params: 4,385		
Non-trainable params: 0		

Figure 4-47. Code to create the neural network

Compile the model using RMSprop as the optimizer and mean squared error for the loss computation. The RMSprop optimizer is similar to the gradient descent algorithm with momentum. Figure 4-48 shows the code to compile the model.

```
autoencoder.compile(optimizer=RMSprop(),
                    loss='mean_squared_error',
                    metrics=['mae', 'accuracy'])
```

Figure 4-48. Code to compile the model

Now, you can start training the model using the training dataset to validate the model at every step. Choose 32 as the batchsize and 20 epochs. The training process outputs the loss and accuracy as well as the validation loss and validation accuracy at each epoch. Figure 4-49 shows the code to start training the model.

```
batch_size = 32
epochs = 20

history = autoencoder.fit(x_train_noisy, x_train,
                        batch_size=batch_size,
                        epochs=epochs,
                        verbose=1,
                        shuffle=True,
                        validation_data=(x_test_noisy, x_test),
                        callbacks=[TensorBoard(log_dir='../logs/{0}'.format(logfilename))])
```

Train on 60000 samples, validate on 10000 samples

Epoch	Train Samples	Time	Loss	Acc	Val Loss	Val Acc
Epoch 1/20	60000/60000	13s 212us/step	0.0370	0.8005	0.0257	0.8007
Epoch 2/20	60000/60000	12s 200us/step	0.0229	0.8072	0.0197	0.8089
Epoch 3/20	60000/60000	12s 198us/step	0.0200	0.8091	0.0190	0.8107
Epoch 4/20	60000/60000	12s 193us/step	0.0185	0.8100	0.0170	0.8092
Epoch 5/20	60000/60000	12s 193us/step	0.0176	0.8105	0.0168	0.8111
Epoch 6/20	60000/60000	12s 192us/step	0.0170	0.8108	0.0163	0.8114
Epoch 7/20	60000/60000	11s 191us/step	0.0164	0.8111	0.0159	0.8110
Epoch 8/20	60000/60000	12s 193us/step	0.0160	0.8113	0.0156	0.8094
Epoch 9/20	60000/60000	11s 189us/step	0.0157	0.8115	0.0155	0.8109
Epoch 10/20	60000/60000	11s 190us/step	0.0153	0.8117	0.0147	0.8099
Epoch 11/20	60000/60000	12s 203us/step	0.0151	0.8118	0.0142	0.8108
Epoch 12/20	60000/60000	12s 192us/step	0.0149	0.8119	0.0144	0.8099
Epoch 13/20	60000/60000	12s 192us/step	0.0147	0.8120	0.0166	0.8124
Epoch 14/20	60000/60000	12s 192us/step	0.0145	0.8121	0.0149	0.8123
Epoch 15/20	60000/60000	11s 190us/step	0.0144	0.8121	0.0134	0.8109
Epoch 16/20	60000/60000	11s 191us/step	0.0143	0.8122	0.0133	0.8110
Epoch 17/20	60000/60000	11s 190us/step	0.0141	0.8122	0.0140	0.8101
Epoch 18/20	60000/60000	11s 191us/step	0.0140	0.8123	0.0153	0.8088
Epoch 19/20	60000/60000	12s 192us/step	0.0139	0.8123	0.0135	0.8107
Epoch 20/20	60000/60000	11s 190us/step	0.0138	0.8124	0.0127	0.8117

Figure 4-49. Code to start training the model

Now that the training process is complete, let's evaluate the model for loss and accuracy. Figure 4-50 shows that the accuracy is 0.81, which is pretty good. It also shows the code to evaluate the model.

```
score = autoencoder.evaluate(x_test, x_test, verbose=1)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

10000/10000 [=====] - 1s 68us/step
Test loss: 0.010875144922733306
Test accuracy: 0.8120423462867736
```

Figure 4-50. Code to evaluate the model

The next step is to use the model to generate the output images for the testing subset. This will show you how well the reconstruction phase is going on. Figure 4-51 shows the code to display denoised images.

```
decoded_imgs = autoencoder.predict(x_test_noisy)

n = 10
plt.figure(figsize=(20, 4))
for i in range(1, n):
    # display original
    ax = plt.subplot(2, n, i)
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(2, n, i + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

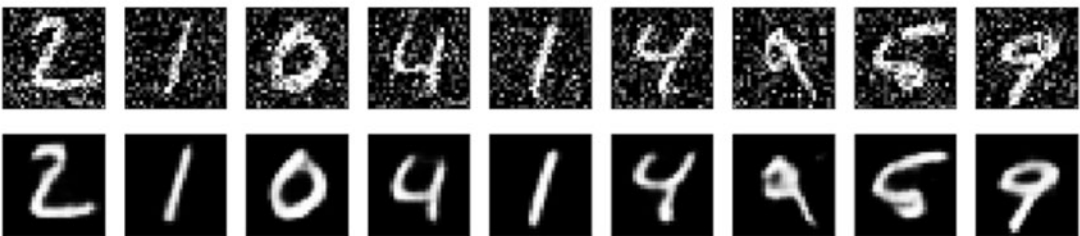


Figure 4-51. Code to display denoised images

You can also see how the encoder phase is working by displaying the test subset images in this phase. Figure 4-52 show the code to display encoded images.

```
encoder = Model(input_img, encoded)
encoded_imgs = encoder.predict(x_test_noisy)
n = 10
plt.figure(figsize=(20, 8))
for i in range(1, n):
    ax = plt.subplot(1, n, i)
    plt.imshow(encoded_imgs[i].reshape(4, 4 * 8).T)
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

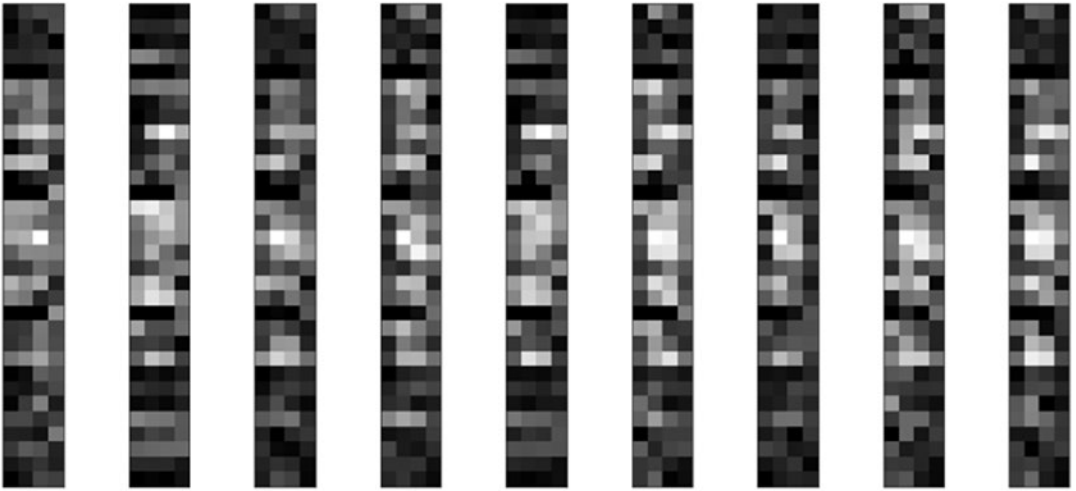
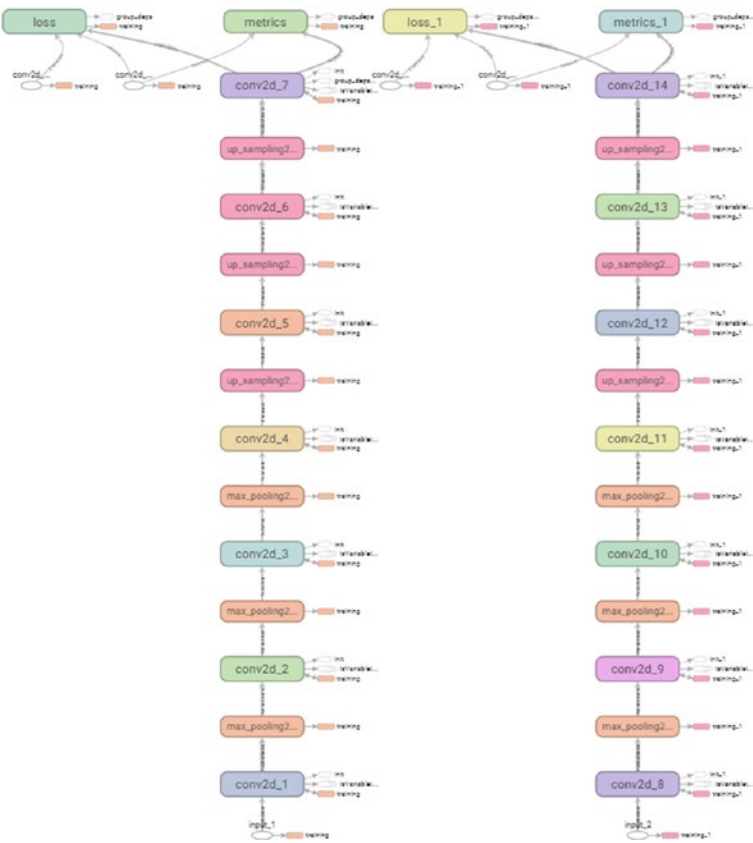


Figure 4-52. Code to display encoded images

Figure 4-53 shows the graph of the model as visualized by TensorBoard.

Main Graph



Auxiliary Nodes

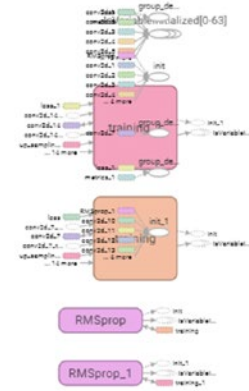


Figure 4-53. Model graph shown in TensorBoard

Figure 4-54 shows the plotting of the accuracy during the training process through the epochs of training.

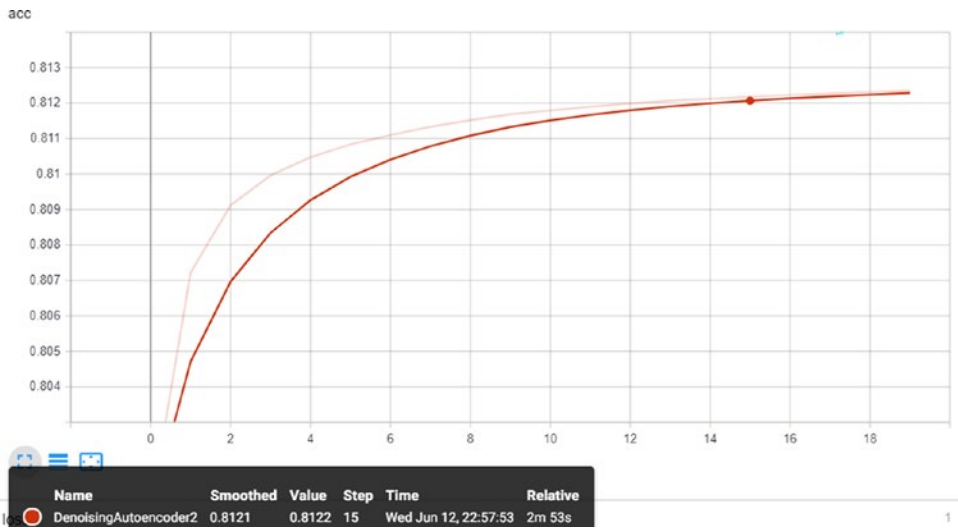


Figure 4-54. Plotting of accuracy shown in TensorBoard

Figure 4-55 shows the plotting of the loss during the training process through the epochs of training.

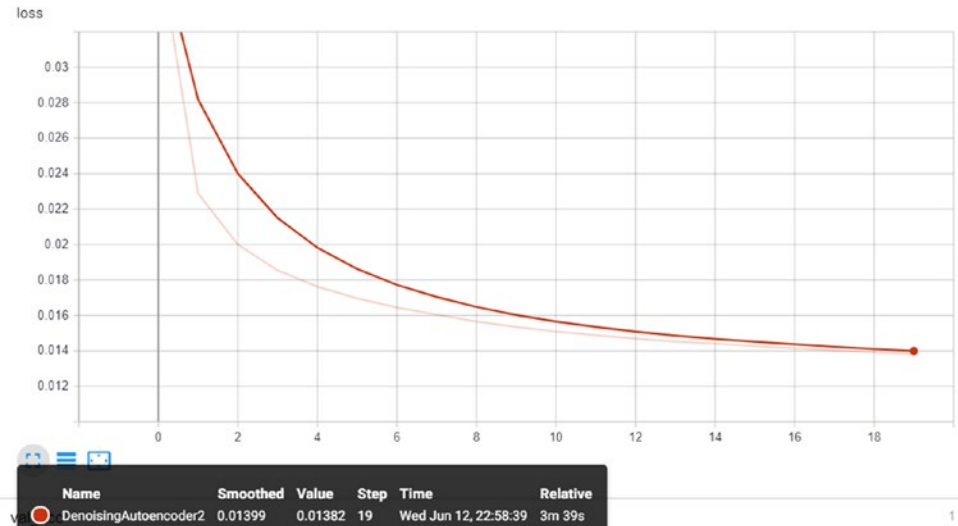


Figure 4-55. Plotting of loss shown in TensorBoard

Figure 4-56 shows the plotting of the accuracy of validation during the training process through the epochs of training.

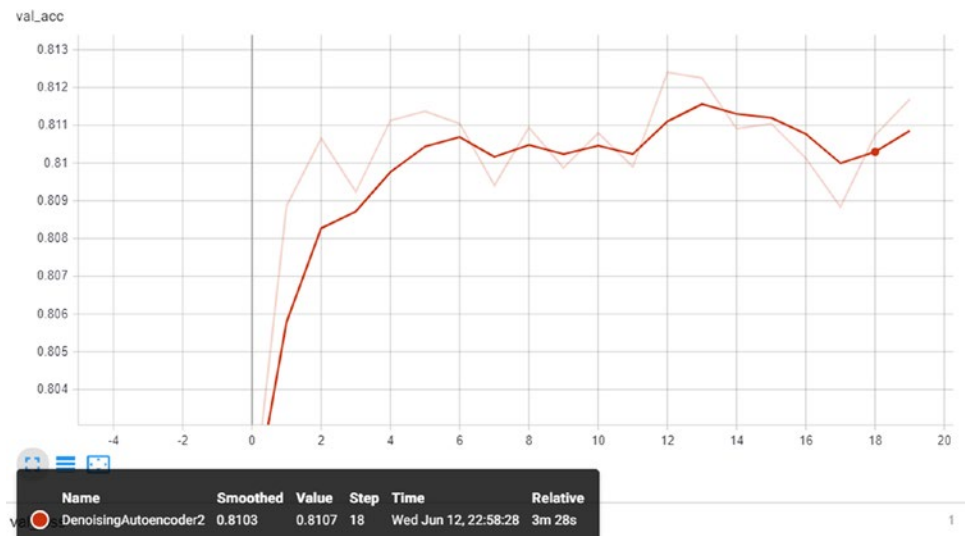


Figure 4-56. *Plotting of validation accuracy shown in TensorBoard*

Figure 4-57 shows the plotting of the loss of validation during the training process through the epochs of training.

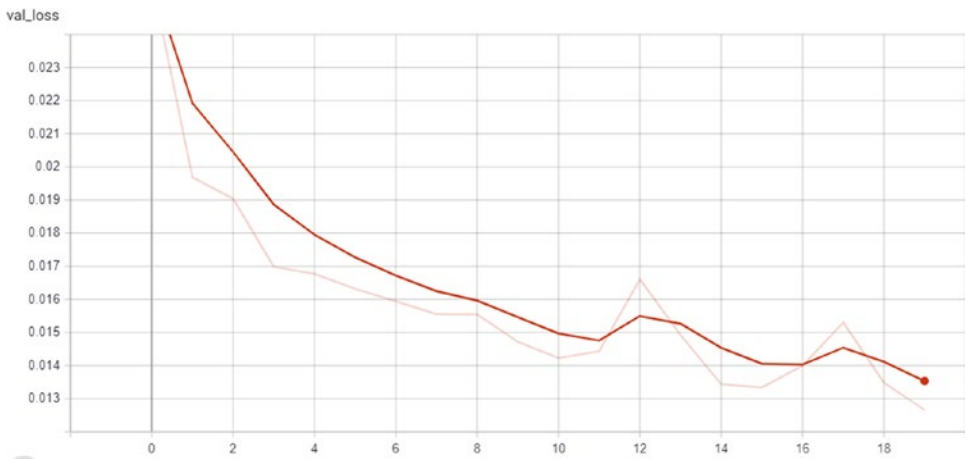


Figure 4-57. *Plotting of validation loss shown in TensorBoard*

Variational Autoencoders

A variational autoencoder is a type of autoencoder with added constraints on the encoded representations being learned. More precisely, it is an autoencoder that learns a latent variable model for its input data. So instead of letting your neural network learn an arbitrary function, you learn the parameters of a probability distribution modeling your data. If you sample points from this distribution, you can generate new input data samples. This is the reason why variational autoencoders are considered to be generative models.

Essentially, VAEs attempt to make sure that encodings that come from some known probability distribution can be decoded to produce reasonable outputs, **even if they are not encodings of actual images**.

In many real-world use cases, we have a whole bunch of data that we're looking at it (it could be images, it could be audio or text; well, it could be anything) but the underlying data that needs to be processed might be lower in dimensions than the actual data, so lot of the machine learning models involve some sort of dimensionality reduction. One very popular technique is singular value decomposition or principal component analysis. Similarly, in the deep learning space, variational autoencoders do the task of reducing the dimensions.

Before we dive into the mechanics of variational autoencoders, let's just recap the normal autoencoders that you saw in this chapter. Autoencoders basically use an encoder and decoder layer at a minimum to reduce the input data features into a latent representation by the encoder layer. The decoder expands the latent representation to generate the output with the goal of training the model well enough to reproduce the input as the output. Any discrepancy between the input and output could signify some sort of abnormal behavior or deviation from what is normal, otherwise known as anomaly detection. In a way, the output gets compressed into a smaller representation but has less dimension than the input, and this is what we call the bottleneck. From the bottleneck, we try to reconstruct the input.

Now that you have the basic concept of the normal autoencoders, let's look at the variational autoencoders. In variational autoencoders, instead of mapping the input to a fixed vector, we map the input to a distribution so the big difference is that the bottleneck vector seen in the normal order in quarters is replaced with the mean vector and a standard deviation vector by looking at the distributions and then taking the sampled latent vector as the actual bottleneck. Clearly this is very different from the normal autoencoder where the input directly yields a latent vector.

First, an encoder network turns the input sample x into two parameters in a latent space, which you can call z_mean and z_log_sigma . Then, you randomly sample similar points z from the latent normal distribution that is assumed to generate the data, via $z = z\_mean + \exp(z\_log\_sigma) * \epsilon$, where ϵ is a random normal tensor. Finally, a decoder network maps these latent space points back to the original input data. Figure 4-58 depicts the variational encoder neural network.

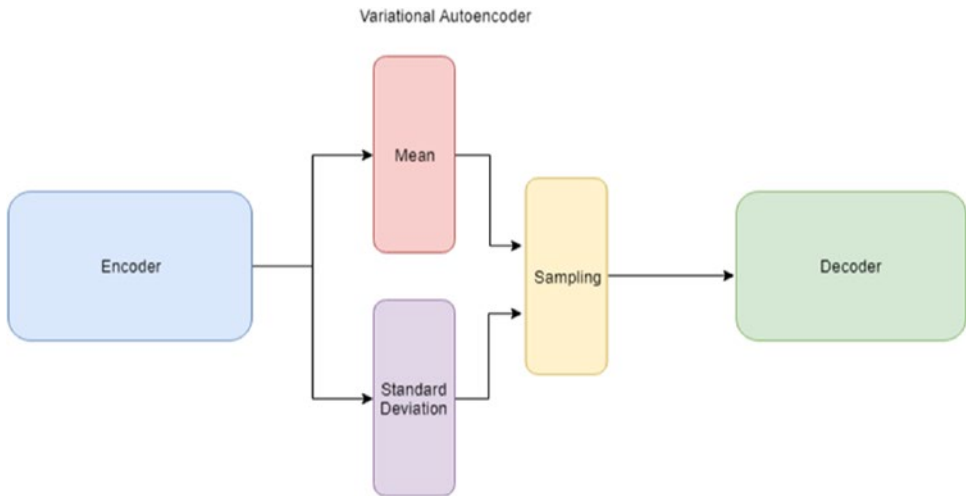


Figure 4-58. *The variational encoder neural network*

The parameters of the model are trained via two loss functions: a reconstruction loss forcing the decoded samples to match the initial inputs (just like in the previous autoencoders), and the KL divergence between the learned latent distribution and the prior distribution, acting as a regularization term. You can actually get rid of this latter term entirely, although it does help in learning well-formed latent spaces and reducing overfitting to the training data.

The distribution that you're learning from is not too far removed from a normally distributed so you going to try to force your latent distribution to be relatively close to a mean of zero and a standard deviation of one so before you can train your variational autoencoder you must consider that there is a sampling problem that could happen. Since you are only taking a sample of the distribution from the mean vector and the standard deviation, it is harder to realize backpropagation there. You are sampling it so how do you get back during the back propagation step?

A variational autoencoder is a kind of a mix of neural networks and graphical models since the first paper came up on variational autoencoder tried to create a graphical model and then turn the graphical model to a neural network. The variational auto encoder is based on variational inference.

Assume that there are two different distributions, p and q , and that you can use KL divergence to show dissimilarity between the two distributions, p and q . Thus, a KL divergence serves as a measure of the similarity between the two distributions, p and q .

The best way to understand the need for a variational autoencoder is that in a general autoencoder, the bottleneck is too dependent on the inputs and there is no understanding of the nature of the data. Since you use sampling of the distribution instead, you will be able to better accommodate the model to new types of data.

Figure 4-59 shows the basic code to import all necessary packages in Jupyter. Also note the versions of the various necessary packages.

```

import keras
from keras import optimizers
from keras import losses
from keras import backend as K
from keras.models import Sequential, Model
from keras.layers import Lambda, Dense, Input, Dropout, Embedding, LSTM
from keras.optimizers import RMSprop, Adam, Nadam
from keras.preprocessing import sequence
from keras.callbacks import TensorBoard
from keras.losses import mse, binary_crossentropy

import sklearn
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, roc_auc_score
from sklearn.preprocessing import MinMaxScaler

import seaborn as sns
import pandas as pd
import numpy as np
import matplotlib

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
%matplotlib inline

import tensorflow
import sys
print("Python: ", sys.version)

print("pandas: ", pd.__version__)
print("numpy: ", np.__version__)
print("seaborn: ", sns.__version__)
print("matplotlib: ", matplotlib.__version__)
print("sklearn: ", sklearn.__version__)
print("Keras: ", keras.__version__)
print("Tensorflow: ", tensorflow.__version__)

```

```

Python: 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)]
pandas: 0.24.2
numpy: 1.16.3
seaborn: 0.9.0
matplotlib: 3.0.3
sklearn: 0.20.3
Keras: 2.2.4
Tensorflow: 1.13.1

```

Figure 4-59. Code to import packages in Jupyter

Figure 4-60 shows the code to visualize the results via a confusion matrix, a chart for the anomalies, and a chart for the errors (difference between predicted and truth) while training.

```

class Visualization:
    labels = ["Normal", "Anomaly"]

    def draw_confusion_matrix(self, y, ypred):
        matrix = confusion_matrix(y, ypred)

        plt.figure(figsize=(10, 8))
        colors = ["orange", "green"]
        sns.heatmap(matrix, xticklabels=self.labels, yticklabels=self.labels, cmap=colors, annot=True, fmt="d")
        plt.title("Confusion Matrix")
        plt.ylabel('Actual')
        plt.xlabel('Predicted')
        plt.show()

    def draw_anomaly(self, y, error, threshold):
        groupsDF = pd.DataFrame({'error': error,
                                'true': y}).groupby('true')

        figure, axes = plt.subplots(figsize=(12, 8))

        for name, group in groupsDF:
            axes.plot(group.index, group.error, marker='x' if name == 1 else 'o', linestyle='',
                      color='r' if name == 1 else 'g', label="Anomaly" if name == 1 else "Normal")

        axes.hlines(threshold, axes.get_xlim()[0], axes.get_xlim()[1], colors="b", zorder=100, label='Threshold')
        axes.legend()

        plt.title("Anomalies")
        plt.ylabel("Error")
        plt.xlabel("Data")
        plt.show()

    def draw_error(self, error, threshold):
        plt.plot(error, marker='o', ms=3.5, linestyle='',
                 label='Point')

        plt.hlines(threshold, xmin=0, xmax=len(error)-1, colors="b", zorder=100, label='Threshold')
        plt.legend()
        plt.title("Reconstruction error")
        plt.ylabel("Error")
        plt.xlabel("Data")
        plt.show()

```

Figure 4-60. Code to visualize the results

You will use the example of credit card data to detect whether a transaction is normal/expected or abnormal/anomaly. Figure 4-61 shows the data being loaded into a Pandas dataframe.

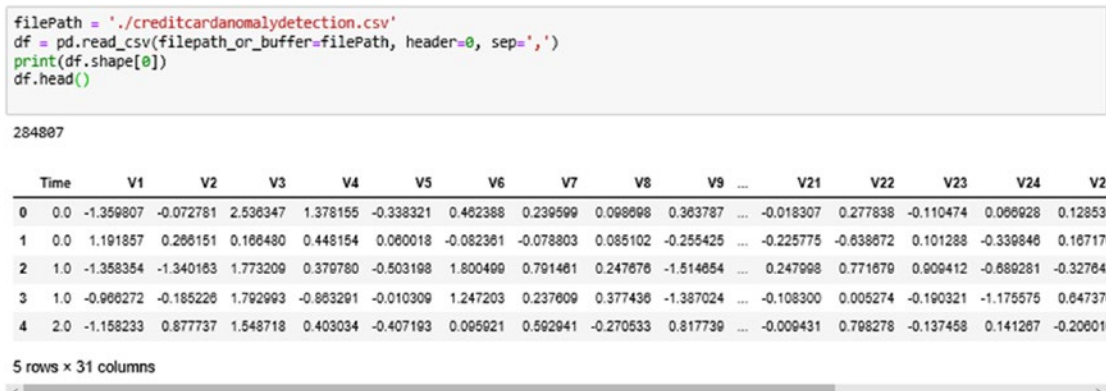


Figure 4-61. Code to load the dataset using Pandas

You will collect 20k normal and 400 abnormal records. You can pick different ratios to try, but in general more normal data examples are better because you want to teach your autoencoder what normal data looks like. Too much of abnormal data in training will train the autoencoder to learn that the anomalies are actually normal, which goes against your goal. Figure 4-62 shows the code to take the majority of normal data records with a few abnormal records.



Figure 4-62. Code to take the majority of normal data records with a few abnormal records

Split the dataframe into training and testing data sets (80-20 split). Figure 4-63 shows the code to split the data into train and test subsets.



Figure 4-63. Code to split the data into train and test subsets

The biggest difference between the standard autoencoders you have seen so far and the variational autoencoder is that here you do not just take the inputs as is; rather, you take the distribution of the input data and then sample the distribution. Figure 4-64 shows the code to implement such a sampling strategy.

```
# reparameterization trick
# instead of sampling from Q(z|X), sample epsilon = N(0,I)
# z = z_mean + sqrt(var) * epsilon
def sampling(args):
    """Reparameterization trick by sampling from an isotropic unit Gaussian.
    # Arguments
        args (tensor): mean and log of variance of Q(z|X)
    # Returns
        z (tensor): sampled latent vector
    """
    z_mean, z_log_var = args
    batch = K.shape(z_mean)[0]
    dim = K.int_shape(z_mean)[1]
    # by default, random_normal has mean = 0 and std = 1.0
    epsilon = K.random_normal(shape=(batch, dim))
    return z_mean + K.exp(0.5 * z_log_var) * epsilon
```

Figure 4-64. Code to sample the distributions

Now it's time to create a simple neural network model with an encoder and a decoder phase. You will encode the 29 columns of the input credit card dataset into 12 features using the encoder. The encoder uses the special distribution sampling logic to generate two parallel layers and then wraps the sampling output (above) as a Layer object.

The decoder phase uses this latent vector and reconstructs the input. While doing this, it also measures the error of reconstruction in order to minimize it. Figure 4-65 shows the code to create the neural network.

```

original_dim = x_train.shape[1]

print(original_dim)

input_shape = (original_dim,)
intermediate_dim = 12
batch_size = 32
latent_dim = 2
epochs = 20

# VAE model = encoder + decoder
# build encoder model
inputs = Input(shape=input_shape, name='encoder_input')
x = Dense(intermediate_dim, activation='relu')(inputs)
z_mean = Dense(latent_dim, name='z_mean')(x)
z_log_var = Dense(latent_dim, name='z_log_var')(x)

# use reparameterization trick to push the sampling out as input
# note that "output_shape" isn't necessary with the TensorFlow backend
z = Lambda(sampling, output_shape=(latent_dim,), name='z')([z_mean, z_log_var])

# instantiate encoder model
encoder = Model(inputs, [z_mean, z_log_var, z], name='encoder')
encoder.summary()

# build decoder model
latent_inputs = Input(shape=(latent_dim,), name='z_sampling')
x = Dense(intermediate_dim, activation='relu')(latent_inputs)
outputs = Dense(original_dim, activation='sigmoid')(x)

# instantiate decoder model
decoder = Model(latent_inputs, outputs, name='decoder')
decoder.summary()

# instantiate VAE model
outputs = decoder(encoder(inputs)[2])
vae = Model(inputs, outputs, name='vae_mlp')

# VAE loss = mse_loss or xent_loss + kl_loss
reconstruction_loss = mse(inputs, outputs)

reconstruction_loss *= original_dim
kl_loss = 1 + z_log_var - K.square(z_mean) - K.exp(z_log_var)
kl_loss = K.sum(kl_loss, axis=-1)
kl_loss *= -0.5
vae_loss = K.mean(reconstruction_loss + kl_loss)
vae.add_loss(vae_loss)

```

Figure 4-65. Code to create the neural network

Figure 4-66 shows the code to show the neural network.

29

Layer (type)	Output Shape	Param #	Connected to
encoder_input (InputLayer)	(None, 29)	0	
dense_15 (Dense)	(None, 12)	360	encoder_input[0][0]
z_mean (Dense)	(None, 2)	26	dense_15[0][0]
z_log_var (Dense)	(None, 2)	26	dense_15[0][0]
z (Lambda)	(None, 2)	0	z_mean[0][0] z_log_var[0][0]
Total params: 412 Trainable params: 412 Non-trainable params: 0			
Layer (type)	Output Shape	Param #	
z_sampling (InputLayer)	(None, 2)	0	
dense_16 (Dense)	(None, 12)	36	
dense_17 (Dense)	(None, 29)	377	
Total params: 413 Trainable params: 413 Non-trainable params: 0			

Figure 4-66. Code to show the neural network

Compile the model using adam as the optimizer and mean squared error for the loss computation. Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively based on training data. Figure 4-67 shows the code to compile the model.

```
vae.compile(optimizer='adam',
            loss='mean_squared_error',
            metrics=['accuracy'])
vae.summary()
```

Layer (type)	Output Shape	Param #
encoder_input (InputLayer)	(None, 29)	0
encoder (Model)	[(None, 2), (None, 2), (N 412	
decoder (Model)	(None, 29)	413
Total params: 825 Trainable params: 825 Non-trainable params: 0		

Figure 4-67. Code to compile the model

Now, you can start training the model using the training dataset to validate the model at every step. Choose 32 as the batchsize and 20 epochs. The training process outputs the loss and accuracy as well as the validation loss and validation accuracy at each epoch. Figure 4-68 shows the code to train the model.

```
history = vae.fit(x_train, x_train,
                 batch_size=batch_size,
                 epochs=epochs,
                 verbose=1,
                 shuffle=True,
                 validation_data=(x_test, x_test),
                 callbacks=[TensorBoard(log_dir='../logs/variationalautoencoder1')])
```

WARNING:tensorflow:From C:\ProgramData\Anaconda3\lib\site-packages\tensorflow\python\ops\math_ops.py:3066: to_int32 (from tensorflow.python.ops.math_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
Train on 16320 samples, validate on 4000 samples

Epoch 1/20
16320/16320 [=====] - 3s 199us/step - loss: 50.2655 - acc: 0.1897 - val_loss: 49.6724 - val_acc: 0.2297

Epoch 2/20
16320/16320 [=====] - 3s 164us/step - loss: 46.3044 - acc: 0.2365 - val_loss: 48.8029 - val_acc: 0.2473

Epoch 3/20
16320/16320 [=====] - 3s 164us/step - loss: 45.8263 - acc: 0.2431 - val_loss: 48.5003 - val_acc: 0.2426

Epoch 4/20
16320/16320 [=====] - 3s 163us/step - loss: 45.6072 - acc: 0.2461 - val_loss: 48.3459 - val_acc: 0.2439

Epoch 5/20
16320/16320 [=====] - 3s 160us/step - loss: 45.4739 - acc: 0.2563 - val_loss: 48.2021 - val_acc: 0.2551

Epoch 6/20
16320/16320 [=====] - 3s 159us/step - loss: 45.3394 - acc: 0.2622 - val_loss: 48.0794 - val_acc: 0.2549

Epoch 7/20
16320/16320 [=====] - 3s 161us/step - loss: 45.2370 - acc: 0.2640 - val_loss: 47.9675 - val_acc: 0.2632

Epoch 8/20
16320/16320 [=====] - 3s 157us/step - loss: 45.1393 - acc: 0.2729 - val_loss: 47.9229 - val_acc: 0.2711

Epoch 9/20
16320/16320 [=====] - 3s 169us/step - loss: 45.0666 - acc: 0.2786 - val_loss: 47.8419 - val_acc: 0.2801

Epoch 10/20
16320/16320 [=====] - 3s 200us/step - loss: 44.9606 - acc: 0.2880 - val_loss: 47.6565 - val_acc: 0.2870

Epoch 11/20
16320/16320 [=====] - 3s 172us/step - loss: 44.8787 - acc: 0.2947 - val_loss: 47.5955 - val_acc: 0.2944

Epoch 12/20
16320/16320 [=====] - 3s 175us/step - loss: 44.8350 - acc: 0.3053 - val_loss: 47.5993 - val_acc: 0.2897

Epoch 13/20
16320/16320 [=====] - 3s 203us/step - loss: 44.7879 - acc: 0.3100 - val_loss: 47.5176 - val_acc: 0.3015

Epoch 14/20
16320/16320 [=====] - 3s 174us/step - loss: 44.7441 - acc: 0.3119 - val_loss: 47.4447 - val_acc: 0.3260

Epoch 15/20
16320/16320 [=====] - 3s 175us/step - loss: 44.7091 - acc: 0.3257 - val_loss: 47.4417 - val_acc: 0.3230

Epoch 16/20
16320/16320 [=====] - 3s 177us/step - loss: 44.6967 - acc: 0.3345 - val_loss: 47.3702 - val_acc: 0.3294

Figure 4-68. Code to train the model

Now that the training process is complete, let's evaluate the model for loss and accuracy. Figure 4-69 shows that the accuracy is 0.23. It also shows the code to evaluate the model.

```
score = vae.evaluate(x_test, x_test, verbose=1)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

4080/4080 [=====] - 0s 60us/step
Test loss: 48.38297452739641
Test accuracy: 0.23529411764705882
```

Figure 4-69. Code to evaluate the model

The next step is to calculate the errors, and detect and also plot the anomalies and the errors. Choose a threshold of 10. Figure 4-70 shows the code to predict the anomalies based on the threshold.

```
threshold=10.00
y_pred = vae.predict(x_test)
y_dist = np.linalg.norm(x_test - y_pred, axis=-1)
z = zip(y_dist >= threshold, y_dist)
y_label=[]
error = []
for idx, (is_anomaly, y_dist) in enumerate(z):
    if is_anomaly:
        y_label.append(1)
    else:
        y_label.append(0)
    error.append(y_dist)
```

Figure 4-70. Code to predict the anomalies based on the threshold

Compute the AUC (Area Under the Curve 0.0 to 1.0); it comes up as 0.93, which is very high. Figure 4-71 shows the code to calculate the AUC.

```
roc_auc_score(y_test, y_label)

0.9345736547003569
```

Figure 4-71. Code to calculate AUC

You can now visualize the confusion matrix to see how well you did with the model. Figure 4-72 shows the code to show the confusion matrix.

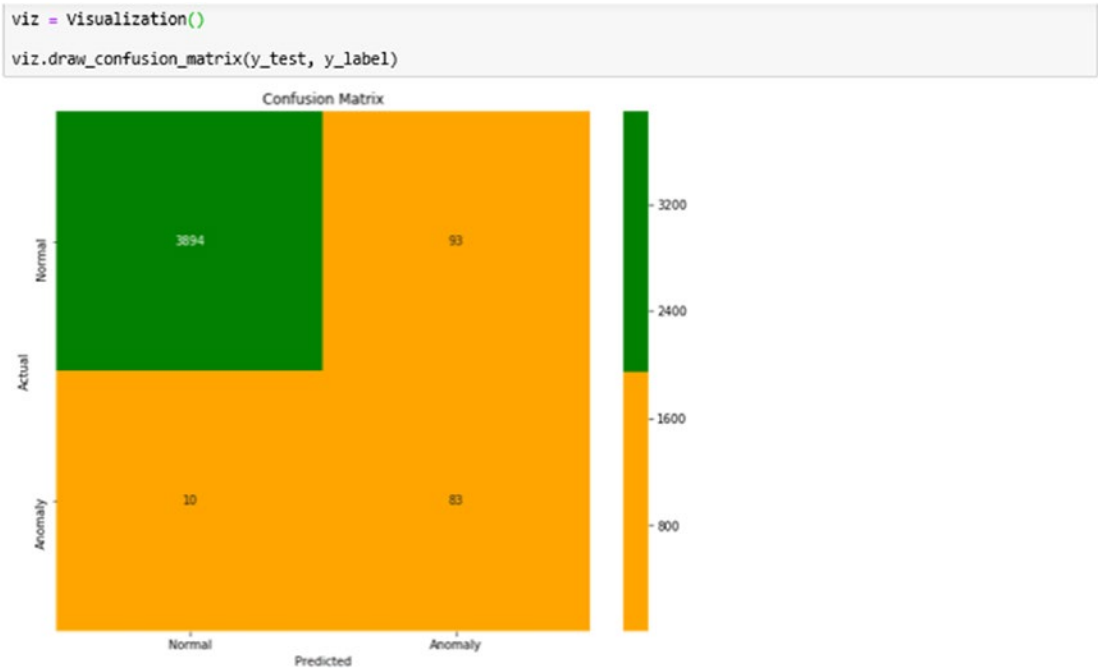


Figure 4-72. Code to show the confusion matrix

Using the predictions of the labels (normal or anomaly) you can plot the anomalies in comparison to the normal data points. Figure 4-73 shows the anomalies relative to the threshold.

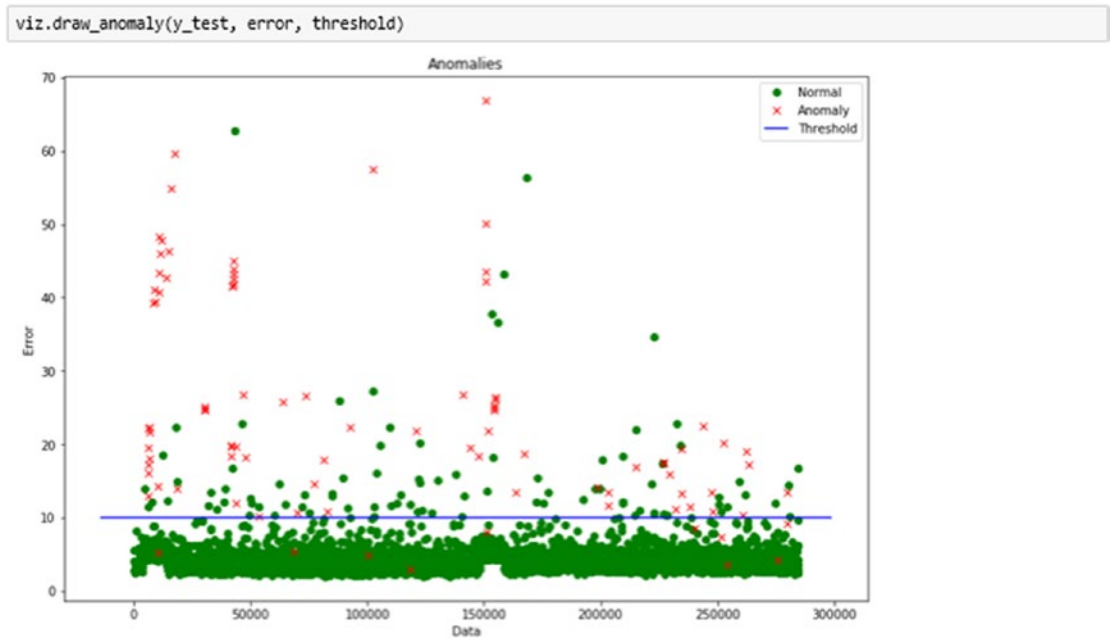


Figure 4-73. Showing the anomalies relative to the threshold

Figure 4-74 shows the graph of the model as visualized by TensorBoard.

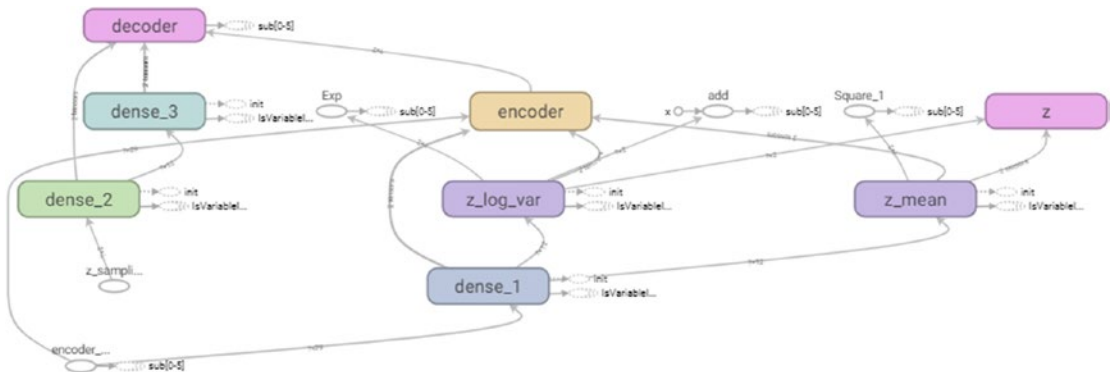


Figure 4-74. Model graph shown in TensorBoard

Figure 4-75 shows the graph of the model as visualized by TensorBoard.

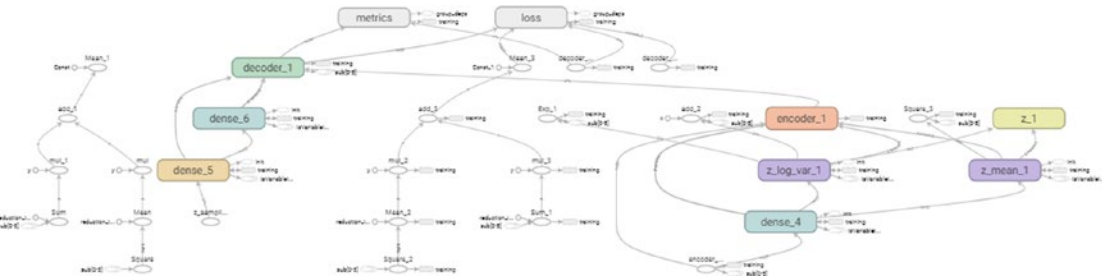


Figure 4-75. Model graph shown in TensorBoard

Figure 4-76 shows the plotting of the accuracy during the training process through the epochs of training.

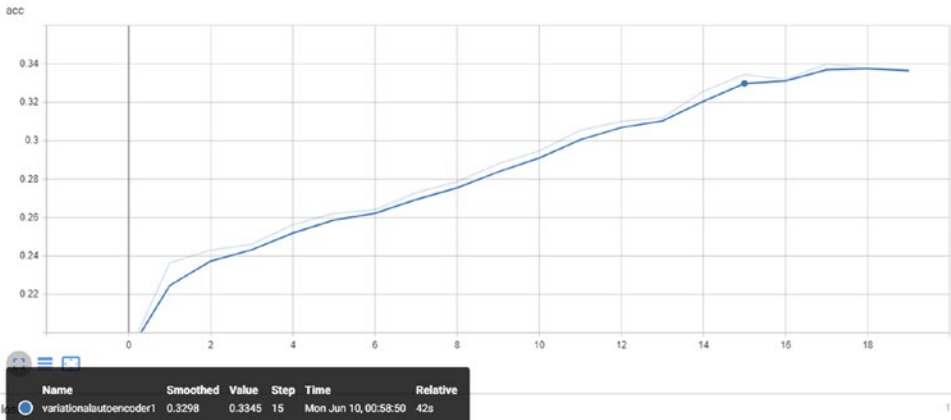


Figure 4-76. Plotting of accuracy shown in TensorBoard

Figure 4-77 shows the plotting of the loss during the training process through the epochs of training.

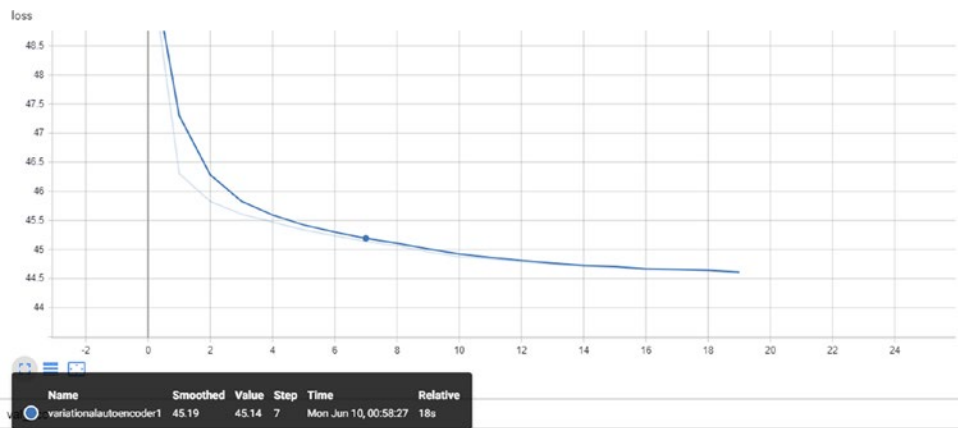


Figure 4-77. Plotting of loss shown in TensorBoard

Figure 4-78 shows the plotting of the accuracy of validation during the training process through the epochs of training.

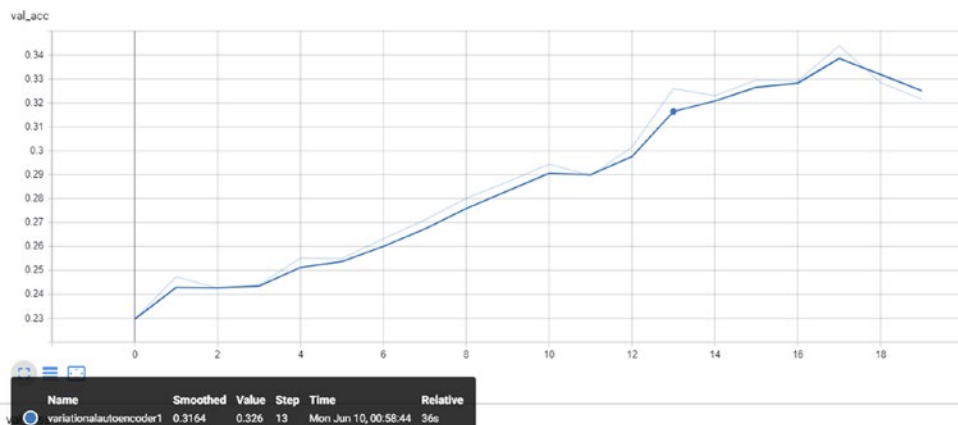


Figure 4-78. Plotting of validation accuracy shown in TensorBoard

Figure 4-79 shows the plotting of the loss of validation during the training process through the epochs of training.

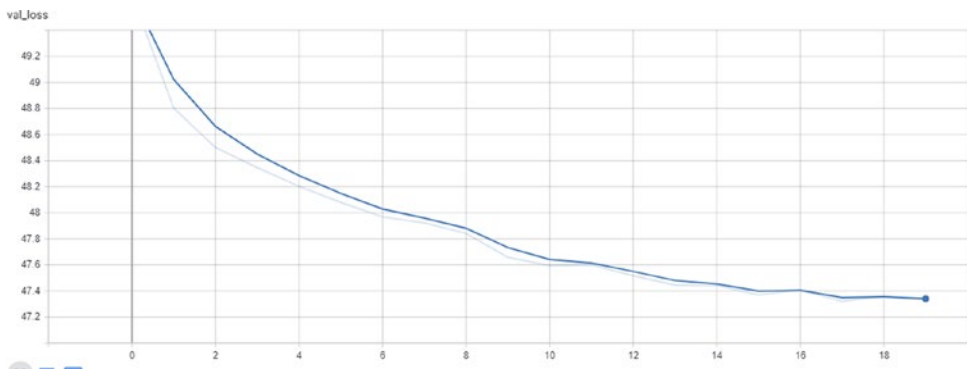


Figure 4-79. Plotting of validation loss shown in TensorBoard

Summary

In this chapter, we discussed autoencoders, types of autoencoders, and how they can be used to build anomaly detection engines. We looked at implementing a simple autoencoder and sparse, deep, convolutional, and denoising autoencoders. We also explored the variational autoencoder as a means to detect anomalies.

In the next chapter, we will look at another method of anomaly detection, **Boltzmann machines**.

CHAPTER 5

Boltzmann Machines

In this chapter, you will learn about Boltzmann machines and how the restricted Boltzmann machine can be used to perform anomaly detection.

In a nutshell, the following topics will be covered throughout this chapter:

- What is a Boltzmann machine?
- Restricted Boltzmann machines (RBMs)
- RBM applications

What Is a Boltzmann Machine?

A **Boltzmann machine** is a special type of bidirectional neural network comprised only of hidden nodes and input nodes, designed to learn the probability distribution of a data set. What makes a Boltzmann machine special is that each and every node is interconnected to each other, meaning the neurons in the hidden layer are connected to each other as well. Additionally, the Boltzmann machine has fixed weights, and the nodes make **stochastic** (probabilistic) decisions about whether or not to fire.

To better understand the model, let's take a look at an example in Figure 5-1.

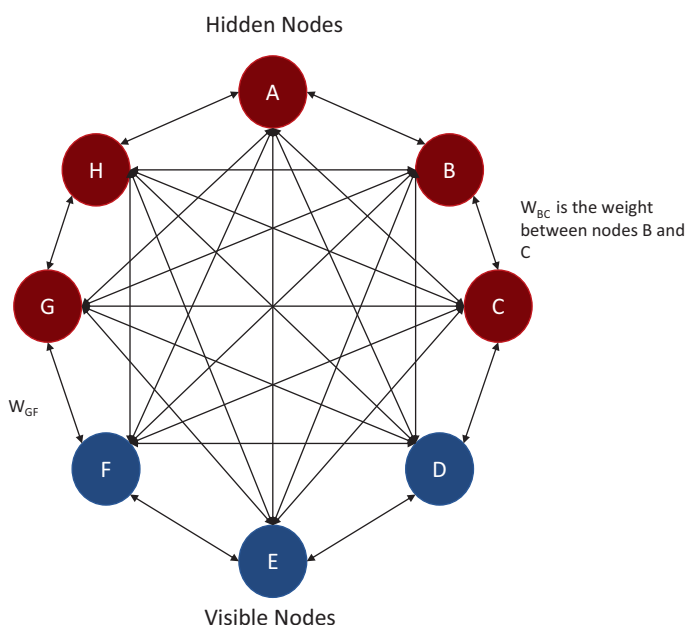


Figure 5-1. A graph showing how a Boltzmann machine can be structured. Notice that all of the nodes are interconnected, even if they are in the same layer

Despite there being a distinction between visible nodes and hidden nodes, that doesn't matter in a Boltzmann machine. In this model, every node communicates to every other node, and the entire model works as a system to create a **generative network** (meaning it's capable of generating its own data based on what it has learned by fitting on a data set). In Boltzmann machines, the visible nodes are what we can interact with; we can't interact with the hidden nodes. One more distinction to make is that there is no training process; the nodes learn to model the data set as best as they can on their own, making the Boltzmann machine an **unsupervised** deep learning model.

However, Boltzmann machines aren't necessarily that practical, and they suffer from problems when the network is scaled up in size. Specific derivations of the Boltzmann machine such as **restricted Boltzmann machines (RBM)**, **deep Boltzmann machines (DBM)**, and **deep belief networks (DBN)** are much more suitable and practical to work with, although they are a bit outdated and have no support from the major frameworks such as Keras, TensorFlow, and PyTorch. Despite that, they still see some new uses today, even though they are overshadowed by newer deep learning models. For our purposes, we will look at applying the **RBM** to anomaly detection, particularly because it is the easiest of the three Boltzmann machine derivations to implement and because it is simpler to work with when we consider the mathematics (which are still at an advanced level) at play.

Restricted Boltzmann Machine (RBM)

The **RBM** is similar to the Boltzmann machine in that it is an unsupervised, stochastic (probabilistic), generative deep learning model. However, a key difference is that the RBM is only comprised of two layers: the input layer and the hidden layer. Its architecture is similar to that of the artificial neural network model you explored in Chapter 3, with the RBM layers looking like the first two layers of an ANN. Because we place a restriction on the layers that none of the nodes within their own layer are to be interconnected, the model is termed as a **restricted** Boltzmann machine. More specifically, since each node outputs a binary value, we are dealing with a **Boolean/Bernoulli RBM**. Figure 5-2 shows an RBM.

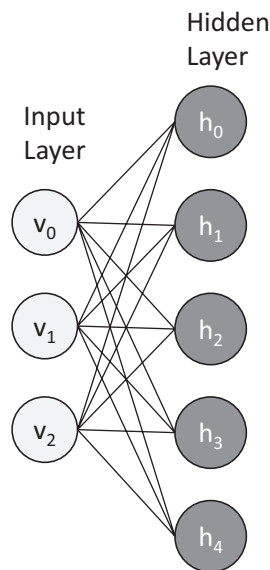


Figure 5-2. A visual representation of a basic restricted Boltzmann machine

We can expand this model out even more to include biases (see Figure 5-3).

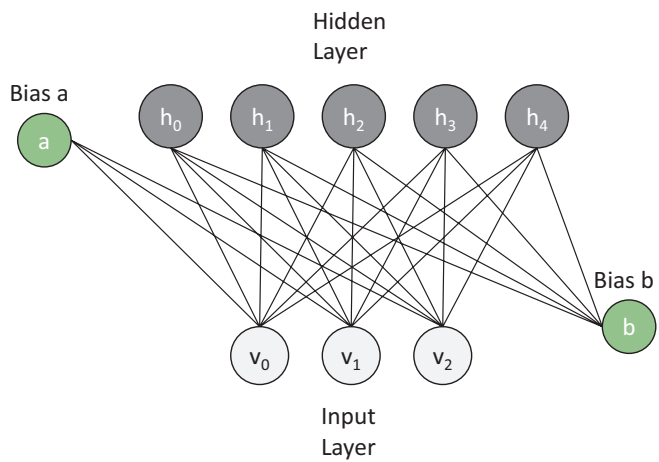


Figure 5-3. A visual representation of a restricted Boltzmann machine with a different bias feeding into each of the two layers

Bias **a** adds to all of the outputs of the input layer, and bias **b** adds to the outputs of the hidden layer. From here, we can define what is called the **energy function**, which the RBM tries to minimize. The **energy function** is shown in Figure 5-4.

$$E(v, h) = - \sum_i a_i v_i - \sum_j b_j h_j - \sum_{i,j} v_i h_j w_{ij}$$

Figure 5-4. A formula that defines the energy function of the restricted Boltzmann machine

The first summation term is an element-wise multiplication between bias **a** and visible layer **v**, where each term a_i is multiplied with each term v_i . The second summation term follows the same logic, except uses element-wise multiplication with bias **b** and hidden layer **h**. Finally, the last summation term multiplies each visible node v_i with each hidden node h_j and the weight value w_{ij} for that connection.

The summations are basically element-wise multiplication between two vectors, one being **transposed**, so **1xn (1 column n rows)**, and the other being **nx1 (n columns 1 row)**. When a vector or matrix is **transposed**, we reverse the dimensions of the vector/matrix and rearrange the values. In a vector, the same values in a row/column are now in a column/row. For matrices, it's a bit more complex. To better understand the concept of **transposing** a vector or matrix, refer to Figure 5-5, Figure 5-6, and Figure 5-7.

Vector vs. Transposed Vector (Figure 5-5)

$$A = \begin{bmatrix} 1 \\ 5 \\ 4 \\ 2 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 5 & 4 & 2 \end{bmatrix}$$

Figure 5-5. Original vector vs. its transposed version**Square Matrix vs. Transposed Square Matrix** (Figure 5-6)

$$B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad B^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Figure 5-6. Original matrix vs. its transposed self. Note how the entries seem to be flipped along the diagonal**Matrix (nxm) vs. Transposed Matrix (mxn)** (Figure 5-7)

$$C = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \quad C^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Figure 5-7. Original nxm matrix vs. its transposed mxn self. The columns of the original matrix C become the rows of the transposed matrix C^T

Rewriting the summations to reflect the multiplication of the respective vectors, one being transposed, the energy function is equivalent to the equation in Figure 5-8.

$$E(v, h) = -a^T v - b^T h - v^T W h$$

Figure 5-8. The equivalent formula for the energy function written without summations

Using the energy function, we can define a **probability function** that will output the probability of the network having a specific **(v,h)**. To elaborate on **v** and **h**, **v** is a vector that represents the states of each node in the input layer, and **h** is a vector that represents the states of each node in the hidden layer.

The **probability function** is shown in Figure 5-9, given a specific (v,h).

$$p(v, h) = \frac{1}{Z} e^{-E(v, h)}$$

Figure 5-9. The probability function that is associated with the visible layer v and the hidden layer h

Z is defined as shown in Figure 5-10.

$$Z = \sum_{v, h} e^{-E(v, h)}$$

Figure 5-10. Z performs the operation over every possible v and h in the data set, so you can see how it forms a probability function. (Say you want a probability of all hearts in a card deck. This is 13/52, with 13 being all of the hearts and 52 being the total number of cards.)

Z is the sum of the function $e^{-E(v, h)}$ over every single pair of input and hidden layer state vectors (a vector representing the states of the layer). The parameters passed into $p(v, h)$ are supposed to be vectors representing a specific configuration of the two layers in terms of what neurons are activated.

You can see how this forms a probability function, since we want to find $e^{-E(v, h)}$ for some $\mathbf{v}$, $\mathbf{h}$ over the sum of $e^{-E(v, h)}$ for all possible pairs of $\mathbf{v}$, $\mathbf{h}$.

We can go a step further and define formulas for the probability of $\mathbf{v}$ or $\mathbf{h}$ given $\mathbf{h}$ or $\mathbf{v}$ (see Figure 5-11 and Figure 5-12).

$$p(h|v) = \frac{p(h, v)}{p(v)} = \prod_j^m p(h_j, v)$$

Figure 5-11. Formula for the probability of the hidden layer being in the state h given the visible layer being in the state v

$$p(v|h) = \frac{p(v, h)}{p(h)} = \prod_i^n p(v_i, h)$$

Figure 5-12. Formula for the probability of the hidden layer being in the state v given the visible layer being in the state h

The Π works similarly to Σ , except with multiplication instead of addition. Essentially, $p(\mathbf{h} | \mathbf{v})$ is the multiplication of every $p(h_i, \mathbf{v})$ that exists. In these cases, $\mathbf{m}$ is the number of hidden nodes, and $\mathbf{n}$ is the number of visible nodes.

This could be a bit complex, so just know that the formulas in Figure 5-11 and 5-12 are basically to find the probabilities of $\mathbf{v}$ or $\mathbf{h}$ being in their states given their respective $\mathbf{h}$ or $\mathbf{v}$ layer counterparts.

From there, we can define two more formulas regarding the probability that a particular node v_i or h_j activates given the vector $\mathbf{h}$ or $\mathbf{v}$, respectively (see Figure 5-13 and Figure 5-14).

$$p(v_i = 1 | \mathbf{h}) = \sigma(a_i + \sum_{j=1}^m w_{ij} h_j)$$

Figure 5-13. The probability of one particular node v_i activating given the multiplication of the weights between v_i and every single hidden node added with the bias

$$p(h_j = 1 | \mathbf{v}) = \sigma(b_j + \sum_{i=1}^n w_{ij} v_i)$$

Figure 5-14. The probability of one particular node h_j activating given the multiplication of the weights between h_j and every single visible node added with the bias

The σ represents the sigmoid function, defined by the formula in Figure 5-15.

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

Figure 5-15. The formula for a sigmoid function

Finally, given training inputs, we want to maximize the joint probability of the inputs, given by the formula in Figure 5-16.

$$\arg \max_W \prod_{v \in V} p(v)$$

Figure 5-16. We are maximizing the joint probability of every possible visible node (the inputs) with respect to the weights

Essentially, we will end up with a huge chain of multiplication of $p(\mathbf{v})$ for every possible $\mathbf{v}$ given $\mathbf{V}$, the set of all possible training inputs. We take that, and we want to maximize that product with respect to the weights $\mathbf{W}$, so we want the weights to be increasing the joint probability (that product of all possible $\mathbf{v}$ layers).

We can also rewrite this in terms of maximizing the expected value of the log probability as shown in Figure 5-17.

$$\arg \max_{\mathbf{W}} E \left[\sum_{v \in V} \log p(v) \right]$$

Figure 5-17. We take the log of $p(v)$ for some v that's a part of the whole training set $\mathbf{V}$. Then we sum those terms up (think back to the log rules) and find the average of them all. That is what we want to maximize with respect to the weights $\mathbf{W}$

The notation $E []$ stands for the **expected value**. In probability, $E(X)$ is the expected value of some random variable X and can be thought of as the **mean**. In our case, we are trying to maximize the mean value of the log probability. Once again, $\mathbf{V}$ is the set of all training inputs.

So to explain what the formula means, we use log rules to rewrite the joint probability as a summation instead, and then we seek to maximize the average of that sum with respect to $\mathbf{W}$, the weights. We want to adjust the weights so that we continue to maximize this expected value for every input in the entire training set.

The formulas pertaining to the RBM can get more complicated and detailed, but the ones listed so far should hopefully be enough to help you gain a good understanding of what an RBM is and how it works. At its core, the RBM is a probabilistic model that operates in accordance with a set of formulas. Additionally, the goal of the formulas is to help the RBM learn a probability distribution to represent V , explaining why the RBM is an **unsupervised learning** algorithm.

As for the training algorithm, there are two choices: **contrastive divergence (CD)** and **persistent contrastive divergence (PCD)**. These algorithms both use Markov chains to help the training algorithm determine what direction to perform the gradient calculations in, but both differ and have their pros and cons. PCD can get better samples of the data and explore the domain of the input space better, but CD is better at extracting features.

Some RBMs might also incorporate a feature known as **momentum**, which basically allows for an increase in learning speed and can be thought of as simulating a ball rolling down a hill in terms of optimizing the target function. (Think back to gradient descent and how the goal is to get to a local minimum. As the “ball” rolls towards the minimum, it gains “momentum” and descends faster and faster. Once it overshoots, it will gain new momentum in the opposite direction, incentivizing it to reach the minimum faster).

There are more intricacies to the RBM, but in the end, you only need to know that RBMs can be used to create a probability distribution of the input data. We will use this property of RBMs to single out anomalies by checking the probability of that particular sample of occurring.

Anomaly Detection with the RBM - Credit Card Data Set

Now that you know more about the complex mechanisms of the RBM, let’s apply the RBM to a data set and see how it performs. For your application, let’s use the credit card data set, which can be found at www.kaggle.com/mlg-ulb/creditcardfraud/version/3.

Begin by importing all of your packages. For this application, you will only explore how an RBM can be applied to the code, since the source code is quite large. However, you can access the source code through the GitHub link at <https://github.com/aaxwaz/Fraud-detection-using-deep-learning>.

Simply download the folder titled `rbm` and place it in your working directory (wherever you have your notebook file or Python file). In this case, we placed in a folder named `boltzmann_machines`.

Now, import your modules (see Figure 5-18).

```
import pandas as pd
import tensorflow as tf
from sklearn.metrics import roc_auc_score as auc
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from boltzmann_machines.rbm import *

%matplotlib inline
```

Figure 5-18. Importing all the modules you need. `%matplotlib inline` is to save the graph within the Jupyter notebook itself

Next, import the data set.

Run the following (refer to Figure 5-19 for the output):

```
df = pd.read_csv("datasets/creditcardfraud/creditcard.csv", sep=",",
index_col=None, encoding="utf-8-sig")
```

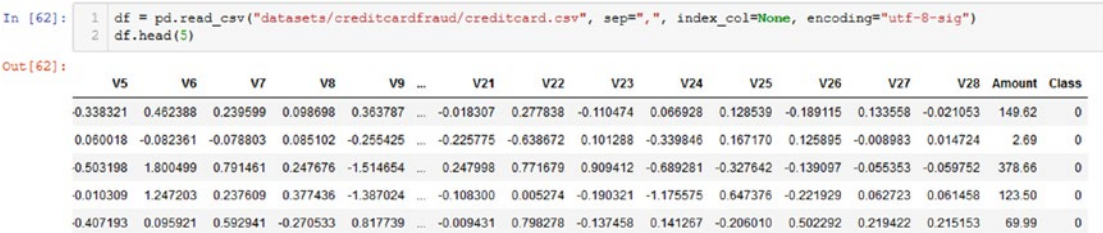


Figure 5-19. Visualizing the data set you just loaded. This figure is scrolled right to show the classes

Looking at the data, it seems that the values in the columns Amount and especially Time need to be normalized. Take a look at how large the values for time get (see Figure 5-20).

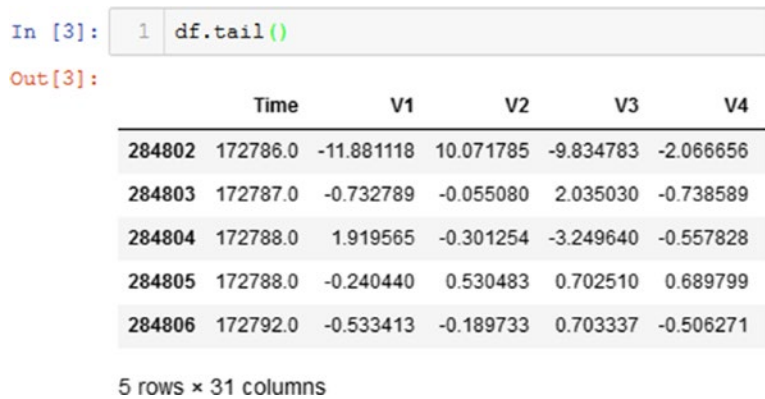


Figure 5-20. Looking at the tail end of the data frame (bottom five entries), the values for time clearly become massive. You must address this in order to train the RBM and ensure that the training process goes smoothly and works properly. Large values like this can ruin the whole process and even lead to no convergence

To avoid numbers like these from potentially ruining the training process, you should standardize the values for both columns. Everything else seems to already be standardized, so you should only worry about these columns. Run the code in Figure 5-21.

```
df['Amount'] =
StandardScaler().fit_transform(df['Amount'].values.reshape(-1, 1))

df['Time'] =
StandardScaler().fit_transform(df['Time'].values.reshape(-1, 1))
```

Figure 5-21. *Standardizing the values in the columns Amount and Time*

Now let's take a look at the values to see how they were transformed (see Figure 5-22 and Figure 5-23).

```
In [5]: 1 df.head()
```

Out[5]:

	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.021053	0.244964	0	
060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	0.014724	-0.342475	0	
503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.059752	1.160686	0	
010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.061458	0.140534	0	
407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	-0.073403	0	

Figure 5-22. *Looking at the values for the column Amount to see how they were standardized*

```
In [6]: 1 df.tail()
```

Out[6]:

	Time	V1	V2	V3	V4	V5
284802	1.641931	-11.881118	10.071785	-9.834783	-2.066656	-5.364473
284803	1.641952	-0.732789	-0.055080	2.035030	-0.738589	0.868229
284804	1.641974	1.919565	-0.301254	-3.249640	-0.557828	2.630515
284805	1.641974	-0.240440	0.530483	0.702510	0.689799	-0.377961
284806	1.642058	-0.533413	-0.189733	0.703337	-0.506271	-0.012546

5 rows × 31 columns

Figure 5-23. *Looking at the values for the column Time to see how they were standardized*

Awesome; looking much better. Now, you can define your training and testing data sets (see Figure 5-24).

```
x_train = df.iloc[:200000, 1:-2].values
y_train = df.iloc[:200000, -1].values

x_test = df.iloc[200000:, 1:-2].values
y_test = df.iloc[200000:,-1].values

print("Shapes:\nx_train:%s\ny_train:%s\n" %
      (x_train.shape, y_train.shape))

print("x_test:%s\ny_test:%s\n" %
      (x_test.shape, y_test.shape))
```

Figure 5-24. This is a different process than usual because of how the RBM model expects the input

You should see something like Figure 5-25 as the output.

```
In [71]: 1
          2 x_train = df.iloc[:200000, 1:-2].values
          3 y_train = df.iloc[:200000, -1].values
          4
          5 x_test = df.iloc[200000:, 1:-2].values
          6 y_test = df.iloc[200000:,-1].values
          7
          8 print("Shapes:\nx_train:%s\ny_train:%s\n" % (x_train.shape, y_train.shape))
          9 print("x_test:%s\ny_test:%s\n" % (x_test.shape, y_test.shape))

Shapes:
x_train: (200000, 28)
y_train: (200000,)

x_test: (84807, 28)
y_test: (84807,)
```

Figure 5-25. The output shapes of the training and testing sets

Getting to the model itself, use the code in Figure 5-26.

```
model = RBM(x_train.shape[1], 10, visible_unit_type='gauss',
main_dir='./', model_name='rbm_model.ckpt',
gibbs_sampling_steps=4, learning_rate=0.001, momentum = 0.95,
batch_size=512, num_epochs=20, verbose=1)
```

Figure 5-26. *Initializing the model with a set of parameters*

The parameters are as follows:

- **num_visible:** The number of nodes in the visible layer
- **num_hidden:** The number of nodes in the hidden layer
- **visible_unit_type:** If the visible units are of type binary or gauss
- **main_dir:** The main directory where to put the models and the directories for data and summary
- **model_name:** The name of the model used when saving
- **gibbs_sampling_steps:** (Optional) Default is 1.
- **learning_rate:** (Optional) Set to the default value of 0.01. Specifies the learning rate.
- **momentum:** The value for momentum to use in gradient descent. Default is 0.9.
- **l2:** The l2-weight decay. Default is 0.001.
- **batch_size:** (Optional) Default is 10.
- **num_epochs:** (Optional) Default is 10.
- **stddev:** (Optional) Default is 0.1. Ignored if the visible_unit_type is not gauss.
- **verbose:** (Optional) Default is 0. A value of 1 shows the outputs, and 0 shows nothing.
- **plot_training_loss:** Whether or not to plot the training loss. Default is True.

Now you can fit the data to the model. Run the following (refer to Figure 5-27 for the output):

```
model.fit(x_train, validation_set=x_test)
```

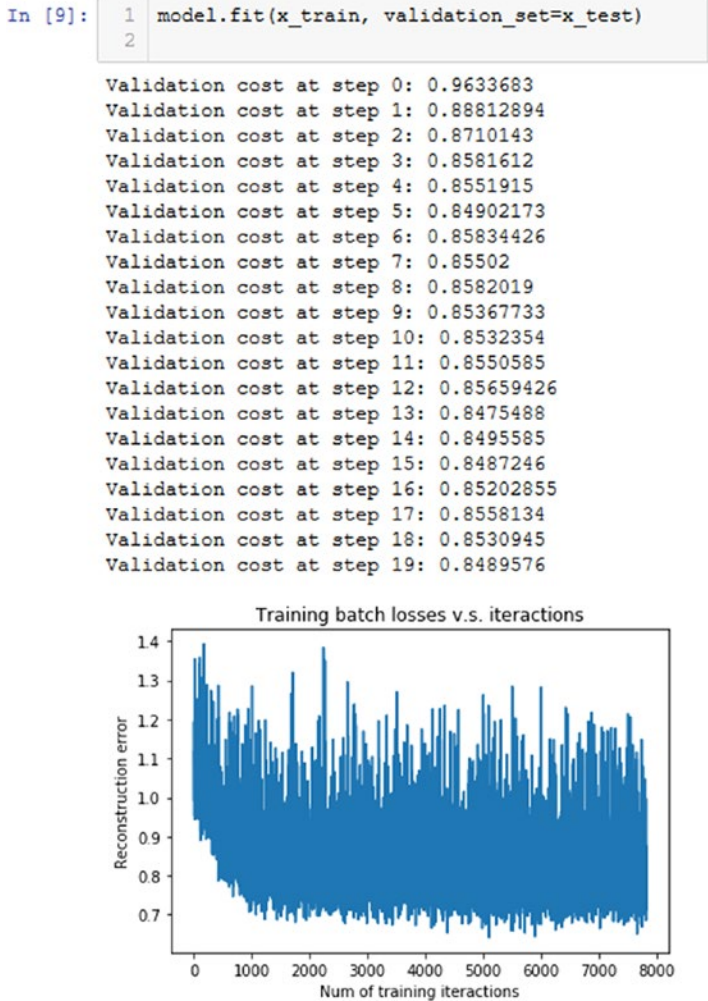


Figure 5-27. The output of training the model

Now that you finished training, you can look at evaluating your model. To get the probability values for each entry in the test set, you have to calculate the free energy for each data point (this is a function unique to this version of the RBM). From there, you can get the probability of each data point occurring given its free energy. Run the code in Figure 5-28.


```
costs = model.getFreeEnergy(x_test).reshape(-1)
score = auc(y_test, costs)
print("AUC Score: {:.2%}".format(score))
```

Figure 5-28. Code to get the costs from the test set and get the AUC scores from that

The output should be something like Figure 5-29.

```
In [10]: 1 costs = model.getFreeEnergy(x_test).reshape(-1)
          2 score = auc(y_test, costs)
          3 print("AUC Score: {:.2%}".format(score))

INFO:tensorflow:Restoring parameters from ./rbm_model.ckpt
AUC Score: 95.84%
```

Figure 5-29. The AUC score ended up at 95.84%

Considering the seemingly simple architecture of the RBM (with how few nodes there are in the model compared to neural networks), that's a pretty good AUC score!

You can also graph the free energy vs. the probability of each data point to get an idea of what the anomalies look like compared to the normal data points. Before you do that, let's check a five-number summary of each data set to get a sense of how they are distributed.

Figure 5-30 shows the code for the five-number summary of the normal data.

```
normal = pd.DataFrame(costs[y_test==0])
normal.describe()
```

Figure 5-30. Code to check the five-number summary of the normal data

The output should look somewhat like Figure 5-31.

```
In [25]: 1 normal = pd.DataFrame(costs[y_test==0])
          2 normal.describe()
```

```
Out[25]:
```

	0
count	84700.000000
mean	0.760787
std	87.097176
min	-7.088358
25%	-5.302821
50%	-4.028915
75%	-1.369422
max	21804.019531

Figure 5-31. The five-number summary shows that the normal data is right skewed, since the values for each quartile are in the negative, while the outlier values in the tail bring the mean up into the positives

Now let's check the five-number summary of the anomalies (see Figure 5-32).

```
anomaly = pd.DataFrame(costs[y_test==1])
anomaly.describe()
```

Figure 5-32. The code to check the five-number summary for the anomalies

The output should look somewhat like Figure 5-33.

```
In [26]: 1 anomaly = pd.DataFrame(costs[y_test==1])
          2 anomaly.describe()
```

```
Out[26]:
```

	0
count	107.000000
mean	88.472694
std	64.513130
min	-5.289360
25%	36.866241
50%	98.163078
75%	128.187202
max	231.617798

Figure 5-33. Looking at the data, it seems that all of the anomalies are below 250. Knowing this, you can now pick a threshold value so only the relevant data is displayed on the graph

Knowing the general distribution of the data, you can pick a threshold value so that only relevant data is shown on the graph. You know the majority of the normal data is situated around the value zero, so the outliers are irrelevant to you since they won't show up on the graph anyways (a few values for 20,000 won't show up when compared to tens of thousands of values around zero).

And so let's choose a cutoff point of 250, since the maximum free energy for an anomaly is at around 232. Figure 5-34 shows a graph of the free energy vs. the probabilities for the test set.

```
plt.title('Free Energy vs Probabilities for Test Set')
plt.figure(figsize=(15, 10))
plt.xlabel('Free Energy')
plt.ylabel('Probability')
plt.hist(costs[(y_test == 0) & (costs < 250)], bins = 100,
color='green', normed=1.0, label='Normal')
plt.hist(costs[(y_test == 1) & (costs < 250)], bins = 100,
color='red', normed=1.0, label='Anomaly')

plt.legend(loc="upper right")
plt.show()
```

Figure 5-34. Code to plot the free energies associated with `x_test` and the respective probabilities

Figure 5-35 shows the code.

```
In [29]: 1 plt.title('Free Energy vs Probabilities for Test Set')
2 plt.figure(figsize=(15, 10))
3 plt.xlabel('Free Energy')
4 plt.ylabel('Probability')
5 plt.hist(costs[(y_test == 0) & (costs < 250)], bins = 100, color='green', normed=1.0, label='Normal')
6 plt.hist(costs[(y_test == 1) & (costs < 250)], bins = 100, color='red', normed=1.0, label='Anomaly')
7
8 plt.legend(loc="upper right")
9 plt.show()
```

Figure 5-35. The code to graph the free energies of the data points and their probabilities

The output graph is shown in Figure 5-36.

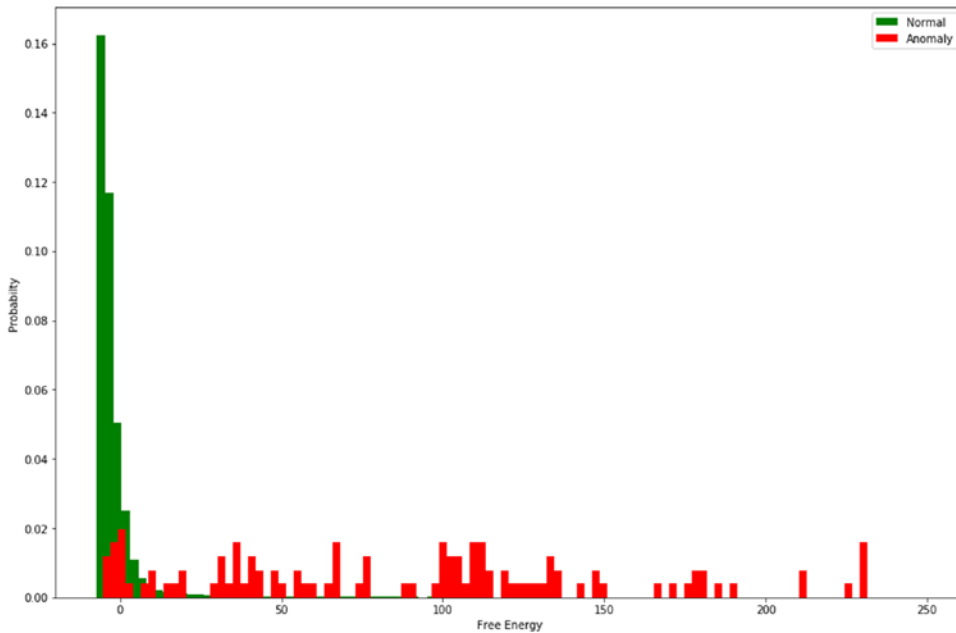


Figure 5-36. *The graph of the free energies vs. the probability of the normal and anomaly data points in the test set with costs less than 500*

The graph automatically graphs the probabilities of the data points based on their free energies, but this isn't exactly made very clear for you to see. The way the probabilities are computed correspond with this line of code:

```
probs = costs / np.sum(costs)
```

This essentially takes the individual free energy and divides it by the total free energy associated with the whole set.

The RBM seems to have learned the distribution well enough that you can see a pretty clear separation between the normal values and the anomalies, although there is a bit of an overlap. In any case, the RBM performed pretty well on the credit card dataset with an AUC of 95.84%.

Anomaly Detection with the RBM - KDDCUP Data Set

Remember the KDDCUP data set you looked at in Chapter 2? Let's try to apply the RBM to it as well. The application will be a similar procedure to that in the previous example, but instead of dealing with excessively large values in the data set, you will learn how to deal with data that is comprised of a hefty number of zero entries.

Again, you begin by importing all of the necessary modules (see Figure 5-37).

```
import pandas as pd
import tensorflow as tf
from sklearn.metrics import roc_auc_score as auc
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from boltzmann_machines.rbm import *
from sklearn.preprocessing import LabelEncoder

%matplotlib inline
```

Figure 5-37. *Importing the necessary modules*

Next, you need to import your data set. Since you've used it before, you don't have to do `df.head()` or print out the shape, but it still helps to get a sense of what the data set looks like (see Figure 5-38).

```
columns = ["duration", "protocol_type", "service", "flag", "src_bytes",
"dst_bytes", "land", "wrong_fragment", "urgent",
        "hot", "num_failed_logins", "logged_in", "num_compromised",
"root_shell", "su_attempted", "num_root",
        "num_file_creations", "num_shells", "num_access_files",
"num_outbound_cmds", "is_host_login",
        "is_guest_login", "count", "srv_count", "error_rate",
"srv_error_rate", "error_rate", "srv_error_rate",
        "same_srv_rate", "diff_srv_rate", "srv_diff_host_rate",
"dst_host_count", "dst_host_srv_count",
        "dst_host_same_srv_rate", "dst_host_diff_srv_rate",
"dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate",
        "dst_host_error_rate", "dst_host_srv_error_rate",
"dst_host_error_rate", "dst_host_srv_error_rate", "label"]

df =
pd.read_csv("datasets/kdd_cup_1999/kddcup.data/kddcup.data.corrected",
sep=",", names=columns, index_col=None)

print(df.shape)

df.head()
```

Figure 5-38. Defining the columns and loading the data set

The output is shown in Figure 5-39.

```
In [321]: 1 columns = ["duration", "protocol_type", "service", "flag", "src_bytes", "dst_bytes", "land", "wrong_fragment", "urgent",
2         "hot", "num_failed_logins", "logged_in", "num_compromised", "root_shell", "su_attempted", "num_root",
3         "num_file_creations", "num_shells", "num_access_files", "num_outbound_cmds", "is_host_login",
4         "is_guest_login", "count", "srv_count", "error_rate", "srv_error_rate", "error_rate", "srv_error_rate",
5         "same_srv_rate", "diff_srv_rate", "srv_diff_host_rate", "dst_host_count", "dst_host_srv_count",
6         "dst_host_same_srv_rate", "dst_host_diff_srv_rate", "dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate",
7         "dst_host_error_rate", "dst_host_srv_error_rate", "dst_host_error_rate", "dst_host_srv_error_rate", "label"]
8
9 df = pd.read_csv("datasets/kdd_cup_1999/kddcup.data/kddcup.data.corrected", sep=",", names=columns, index_col=None)
10 print(df.shape)
11 df.head()

(4898431, 42)

Out[321]:
```

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_srv_count	dst_host_same_srv_rate	dst_host_di
0	0	tcp	http	SF	215	45076	0	0	0	0	...	0	0.0	
1	0	tcp	http	SF	162	4528	0	0	0	0	...	1	1.0	
2	0	tcp	http	SF	236	1228	0	0	0	0	...	2	1.0	
3	0	tcp	http	SF	233	2032	0	0	0	0	...	3	1.0	
4	0	tcp	http	SF	239	486	0	0	0	0	...	4	1.0	

5 rows × 42 columns

Figure 5-39. Notice that there are categorical labels to deal with, and that there are a huge number of columns per data entry

As in Chapter 2, you only want to focus on HTTP attacks, so let's filter the data frame to only include them (see Figure 5-40).

```
df = df[df["service"] == "http"]
df = df.drop("service", axis=1)
columns.remove("service")

print(df.shape)
df.tail()
```

Figure 5-40. Filtering all the entries to include only HTTP attacks and dropping the service column from the data frame

The new output is shown in Figure 5-41.

```
In [358]: 1 df = df[df["service"] == "http"]
          2 df = df.drop("service", axis=1)
          3 columns.remove("service")
          4
          5 print(df.shape)
          6 df.tail()

(623091, 41)
```

Out[358]:

	duration	protocol_type	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logins	...	dst_host_srv_count	dst_host_same_srv_
4898426	0	tcp	SF	212	2288	0	0	0	0	0	...	255	
4898427	0	tcp	SF	219	236	0	0	0	0	0	...	255	
4898428	0	tcp	SF	218	3610	0	0	0	0	0	...	255	
4898429	0	tcp	SF	219	1234	0	0	0	0	0	...	255	
4898430	0	tcp	SF	219	1098	0	0	0	0	0	...	255	

5 rows × 41 columns

Figure 5-41. The columns only consist of HTTP attacks. Here you look at the tail end of the data frame

As a reminder, `df.tail()` performs the same function as `df.head()` but shows the entries from the bottom up as opposed to top down. Also, you can pass a parameter in the parenthesis to indicate the number of rows you want to see.

You don't want values that are strings in your data, so you have to use the label encoder as in Chapter 2 (see Figure 5-42).

```
for col in df.columns:
    if df[col].dtype == "object":
        encoded = LabelEncoder()
        encoded.fit(df[col])
        df[col] = encoded.transform(df[col])

df.head()
```

Figure 5-42. Using the label encoder on the categorical values in your data frame

The new output is shown in Figure 5-43.

```
In [359]: 1 for col in df.columns:
          2     if df[col].dtype == "object":
          3         encoded = LabelEncoder()
          4         encoded.fit(df[col])
          5         df[col] = encoded.transform(df[col])
          6
          7 df.head()

Out[359]:
```

	duration	protocol_type	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logins	...	dst_host_srv_count	dst_host_same_srv_rate	d
0	0	0	9	215	45076	0	0	0	0	0	...	0	0.0	
1	0	0	9	162	4528	0	0	0	0	0	...	1	1.0	
2	0	0	9	236	1228	0	0	0	0	0	...	2	1.0	
3	0	0	9	233	2032	0	0	0	0	0	...	3	1.0	
4	0	0	9	239	486	0	0	0	0	0	...	4	1.0	

5 rows x 41 columns

Figure 5-43. The output showing the new data frame with the categorical values converted to integer label equivalents

In this data set, the normal data entries comprise an overwhelmingly large proportion of the data entries, pretty much drowning out the anomalous data. Not only that, but you don't want to pass in all of the data values into the RBM, so you will create a new data frame that contains a portion of normal data entries and all of the anomalous data entries. Run the code in Figure 5-44.


```

anomalies = df[df["label"] != 4]
normal = df[df["label"] == 4]

for f in range(0, 10):
    normal = normal.iloc[np.random.permutation(len(normal))]

novelties = pd.concat([normal[:50000], anomalies])
novelties.shape

```

Figure 5-44. Code to define an anomaly data set and a normal data set. Then, the normal data set is shuffled to ensure random selections, and a new data set named *novelties* is formed

As in Chapter 2, the normal labels are encoded as 4 so you can use them as the basis to separate the normal entries from the anomalies.

Since the data set is so large, the entries are shuffled randomly ten times before a sample of 50,000 is selected from them. This is to ensure a random selection of values from the entire data set instead of having the entries just in the top 50,000. The output is shown in Figure 5-45.

```

In [360]: 1 anomalies = df[df["label"] != 4]
          2 normal = df[df["label"] == 4]
          3
          4
          5 for f in range(0, 10):
          6     normal = normal.iloc[np.random.permutation(len(normal))]
          7
          8
          9 novelties = pd.concat([normal[:50000], anomalies])
         10 novelties.shape

Out[360]: (54045, 41)

```

Figure 5-45. The output of the code in Figure 5-44

One thing about the KDDCUP data set is that there are a massive amount of entries with data values as either miniscule values or as 0. You've dealt with massive values with the credit card data set, and you know that those values can throw off the training process entirely. Likewise, massive amounts of zero values or really tiny data values can also hamper the training process.

Since `novelties.head()` only displays some of the columns, you'll have to use something else to check every column, so look at the code in Figure 5-46.

```
with pd.option_context('display.max_rows', 5,
    'display.max_columns', 41):
    print(novelties)
```

Figure 5-46. Code to print all the columns and five rows in the data frame

The parameters are self-explanatory. In the example, all 41 columns are displayed for the first 5 rows (Figure 5-47 and Figure 5-48).

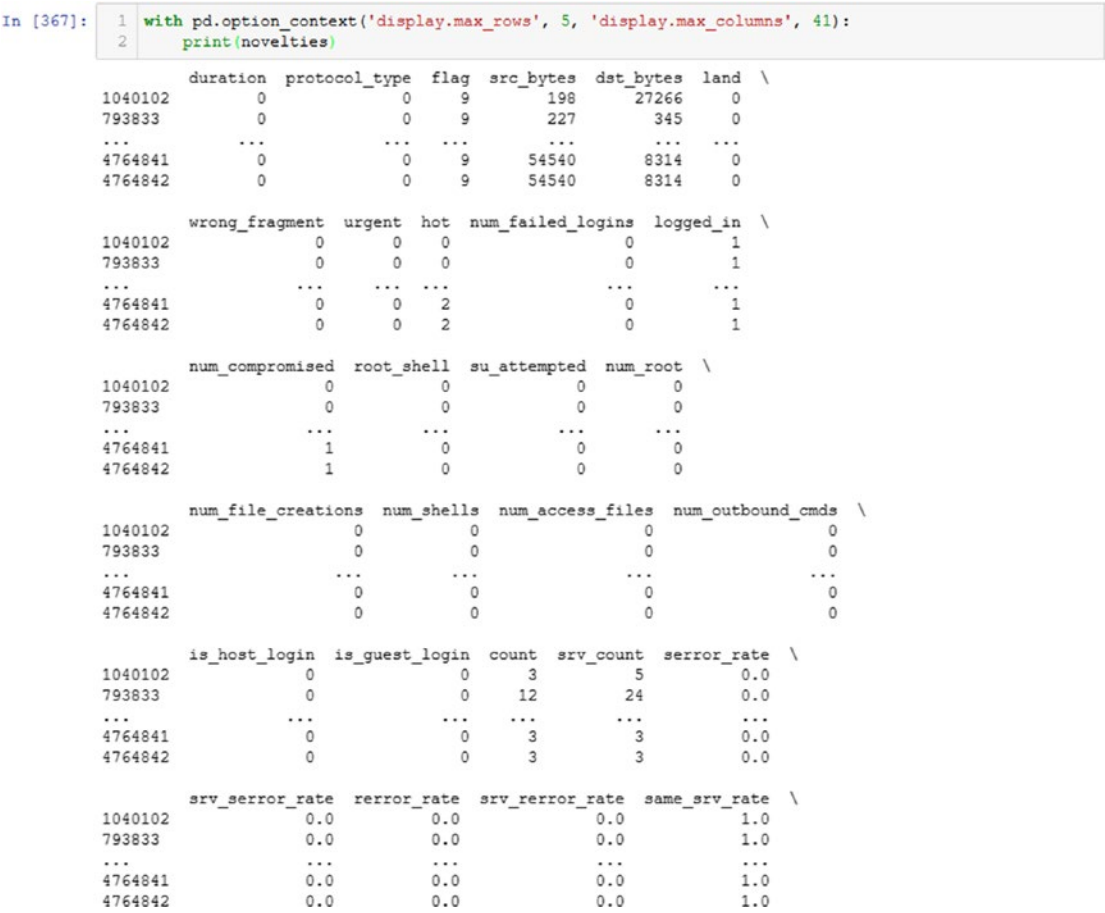


Figure 5-47. The output from the code in Figure 5-46. Notice the massive amount of zero values in the columns of the data entries

```

diff_srv_rate  srv_diff_host_rate  dst_host_count  \
1040102        0.0                0.60            84
793833         0.0                0.12            255
...           ...                ...             ...
4764841        0.0                0.00            99
4764842        0.0                0.00            100

dst_host_srv_count  dst_host_same_srv_rate  dst_host_diff_srv_rate  \
1040102            255                    1.0                0.0
793833            255                    1.0                0.0
...           ...                ...             ...
4764841            99                    1.0                0.0
4764842            100                   1.0                0.0

dst_host_same_src_port_rate  dst_host_srv_diff_host_rate  \
1040102                    0.01                0.03
793833                     0.00                0.00
...           ...                ...             ...
4764841                    0.01                0.00
4764842                    0.01                0.00

dst_host_serror_rate  dst_host_srv_serror_rate  dst_host_rerror_rate  \
1040102              0.01                0.00            0.01
793833              0.00                0.00            0.00
...           ...                ...             ...
4764841              0.01                0.01            0.01
4764842              0.01                0.01            0.01

dst_host_srv_rerror_rate  label
1040102                  0.01      4
793833                   0.00      4
...           ...            ...
4764841                  0.01      0
4764842                  0.01      0

[54045 rows x 41 columns]

```

Figure 5-48. The rest of the output continued from Figure 5-46. There are still many zero values or really small values in each entry

While the large amount of zero-value entries might not have affected the isolation forest, they will certainly mess with the training process of the RBM, leading to terrible AUC scores. Therefore, standardizing all of the values will help the RBM during the training process and help it attain proper AUC scores.

You don't want to standardize the data values for the columns `protocol_type`, `flag`, or `label`, so exclude them specifically (see Figure 5-49).

```
for c in columns:
    if(c != "protocol_type" and c != "flag" and c != "label"):
        novelties[c] =
StandardScaler().fit_transform(novelties[c].values.reshape(-1,
1))

novelties.head()
```

Figure 5-49. Standardizing every value except for the columns the label encoder transformed

The output showing the standardized data is shown in Figure 5-50, Figure 5-51, and Figure 5-52.

```
In [346]: for c in columns:
          if(c != "protocol_type" and c != "flag" and c != "label"):
              novelties[c] = StandardScaler().fit_transform(novelties[c].values.reshape(-1, 1))
          novelties.head()
```

Figure 5-50. The code in a Jupyter cell

Out[346]:

	duration	protocol_type	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logins	...	dst_host_srv_count	dst_host_sai
197882	-0.007301	0	9	-0.199908	-0.179237	0.0	0.0	0.0	-0.209332	0.0	...	0.329176	
369876	-0.007301	0	9	-0.205336	-0.184283	0.0	0.0	0.0	-0.209332	0.0	...	0.329176	
336092	-0.007301	0	9	-0.205336	-0.146532	0.0	0.0	0.0	-0.209332	0.0	...	0.329176	
4789776	-0.007301	0	9	-0.205336	-0.159467	0.0	0.0	0.0	-0.209332	0.0	...	0.329176	
758885	-0.007301	0	9	-0.204213	-0.179466	0.0	0.0	0.0	-0.209332	0.0	...	0.329176	

5 rows x 41 columns

Figure 5-51. The first part of the output showing that most of the values have been transformed

Out[346]:

dst_host_diff_srv_rate	dst_host_same_src_port_rate	dst_host_srv_diff_host_rate	dst_host_serror_rate	dst_host_srv_serror_rate	dst_host_error_rate	dst_host :
-0.124983	0.986873	0.003223	-0.181091	-0.179287	-0.322855	
-0.124983	-0.391602	-0.589299	-0.181091	-0.179287	-0.322855	
-0.124983	-0.391602	-0.391791	-0.181091	-0.179287	-0.322855	
-0.124983	-0.336463	0.200731	-0.181091	-0.179287	-0.286223	
-0.124983	-0.115907	-0.194284	-0.181091	-0.179287	-0.322855	

Figure 5-52. The same output but scrolled right to show that more of the values have been transformed

As you can see, most of the zero value entries have been standardized in accordance with all of the values in their respective columns. The few nonzero entries in these columns will help the scaler to standardize the rest of the values in that column.

Just as you want to avoid massive values in the training set, you also seek to avoid large amounts of zero value entries in the data. In both such cases, the calculations for the gradient will be thrown off, resulting in cases such as the “exploding gradient” (gradients so big that the model can never converge on the local minimum) or the “vanishing gradient” (gradients so small that they are practically nonexistent, and the model never converges on the local minimum). An abundance of values that are too large or too small can negatively affect the training process, so it’s a good idea to preprocess the data set before training the model on it.

Now you can move on to defining your training and testing sets (see Figure 5-53).

```
x_train = novelties.iloc[:43000, 1:-2].values
y_train = novelties.iloc[:43000, -1].values

x_test = novelties.iloc[43000:, 1:-2].values
y_test = novelties.iloc[43000:,-1].values

print("Shapes:\nx_train:%s\ny_train:%s\n" % (x_train.shape,
y_train.shape))
print("x_test:%s\ny_test:%s\n" % (x_test.shape, y_test.shape))

y_test
```

Figure 5-53. *Defining the training and testing sets and printing out the shapes of each*

The corresponding output is shown in Figure 5-54.

```

In [14]: 1 x_train = novelties.iloc[:43000, 1:-2].values
          2 y_train = novelties.iloc[:43000, -1].values
          3
          4 x_test = novelties.iloc[43000:, 1:-2].values
          5 y_test = novelties.iloc[43000:,-1].values
          6
          7
          8 print("Shapes:\nx_train:%s\ny_train:%s\n" % (x_train.shape, y_train.shape))
          9 print("x_test:%s\ny_test:%s\n" % (x_test.shape, y_test.shape))
         10
         11 y_test

Shapes:
x_train: (43000, 38)
y_train: (43000,)

x_test: (11045, 38)
y_test: (11045,)

Out[14]: array([4, 4, 4, ..., 0, 0, 0], dtype=int64)

```

Figure 5-54. The output shapes and some entries of `y_test` are displayed

The 43,000 entries indicate a roughly 80-20 split between the training and testing data sets.

Again, you drop the last column, since this is **unsupervised** training (although it is true that both the anomalies and the normal entries are labeled, the model only sees unlabeled data during the training and prediction processes).

With your data sets created, you can define and train the model (see Figure 5-55, Figure 5-56, and Figure 5-57).

```

model = RBM(x_train.shape[1], 20, visible_unit_type='gauss',
            main_dir='.', model_name='rbm_model2.ckpt',
                gibbs_sampling_steps=4, learning_rate=0.001,
            momentum = 0.95, batch_size=512, num_epochs=20, verbose=1)

```

Figure 5-55. Initializing the model

The code to train the model is shown in Figure 5-56.

```

model.fit(x_train, validation_set=x_test)

```

Figure 5-56. Training the model on `x_train`, using `x_test` as validation data

The output you should see is shown in Figure 5-57.

```
In [17]: 1 model.fit(x_train, validation_set=x_test)
         2
```

```
Validation cost at step 0: 1.4761298
Validation cost at step 1: 1.4645535
Validation cost at step 2: 1.4219579
Validation cost at step 3: 1.4172356
Validation cost at step 4: 1.42167
Validation cost at step 5: 1.4136512
Validation cost at step 6: 1.418542
Validation cost at step 7: 1.3989593
Validation cost at step 8: 1.4185325
Validation cost at step 9: 1.4090425
Validation cost at step 10: 1.4065987
Validation cost at step 11: 1.4020221
Validation cost at step 12: 1.4002018
Validation cost at step 13: 1.4049628
Validation cost at step 14: 1.4142944
Validation cost at step 15: 1.4096367
Validation cost at step 16: 1.3955325
Validation cost at step 17: 1.4014837
Validation cost at step 18: 1.3970937
Validation cost at step 19: 1.3985484
```

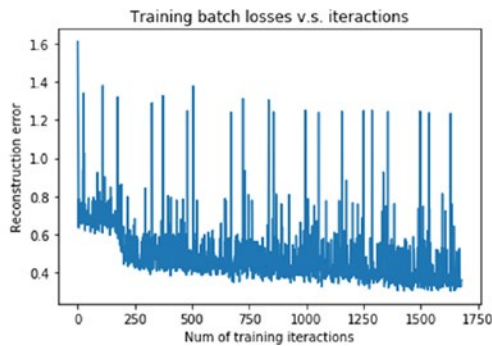


Figure 5-57. The training output by the model for the code in Figure 5-56

Since the labels aren't binary, you want to redefine them as either normal, 0, or anomalous, 1. Run the code in Figure 5-58.

```

for f in range(0, len(y_test)):
    if y_test[f] == 4:
        y_test[f] = 0
    else:
        y_test[f] = 1

y_test

```

Figure 5-58. Code to change all labels that are 4 to 0, representing normal entries, and all labels that aren't 4 to 1, representing anomalies

The output you should see is shown in Figure 5-59.

```

In [353]: 1 for f in range(0, len(y_test)):
          2     if y_test[f] == 4:
          3         y_test[f] = 0
          4     else:
          5         y_test[f] = 1
          6
          7 y_test

Out[353]: array([0, 0, 0, ..., 1, 1, 1], dtype=int64)

```

Figure 5-59. The labels should now be transformed. Some of the entries in `y_test` are shown to make sure they were transformed correctly

Now that your labels have been corrected, you can get the free energy and find the AUC score (see Figure 5-60).

```

costs = model.getFreeEnergy(x_test).reshape(-1)
score = auc(y_test, costs)
print("AUC Score: {:.2%}".format(score))

```

Figure 5-60. Code to get the free energy for each model in `x_test` and then to find the AUC score based on that

The output you should see is shown in Figure 5-61.

```
In [19]: 1 costs = model.getFreeEnergy(x_test).reshape(-1)
          2 score = auc(y_test, costs)
          3 print("AUC Score: {:.2%}".format(score))

INFO:tensorflow:Restoring parameters from ./rbm_model2.ckpt
AUC Score: 99.46%
```

Figure 5-61. The generated AUC score

That's an even better AUC score than for the credit card data set! Let's take a look at what happens when you plot the free energy vs. the probability. As with the previous example, let's take a look at the five-number summary for the normal data to see how the distribution looks (Figure 5-62 and Figure 5-63).

```
normal_data = pd.DataFrame(costs[y_test == 0])
normal_data.describe()
```

Figure 5-62. Code to check the five-number summary of the normal data

The output should look somewhat like Figure 5-63.

```
In [22]: 1 normal_data = pd.DataFrame(costs[y_test == 0])
          2 normal_data.describe()

Out[22]:
```

	0
count	7000.000000
mean	-43.244312
std	22.784908
min	-46.898159
25%	-46.566977
50%	-46.379679
75%	-45.858756
max	1145.513062

Figure 5-63. It seems that the graph is skewed right, and that all of the values are under 1150

Now let's look at the five-number summary to see what the general distribution of the anomalous data looks like (see Figure 5-64 and Figure 5-65).

```
anomalies = pd.DataFrame(costs[y_test == 1])
anomalies.describe()
```

Figure 5-64. Code to check the five-number summary of the anomalous data

The output should look somewhat like Figure 5-65.

```
In [21]: 1 anomalies = pd.DataFrame(costs[y_test == 1])
          2 anomalies.describe()

Out[21]:
```

	0
count	4045.000000
mean	44.125881
std	100.816040
min	-34.099133
25%	-11.051010
50%	-4.358704
75%	89.738434
max	1470.521851

Figure 5-65. Based on the maximum value, you don't need to filter out any values for cost, except for what is an anomaly and what is a normal point

Now you can graph the free energy vs. the probabilities for each value in the test set separated by their label. Run the code in Figure 5-66.

```
plt.title('Free Energy vs Probabilities for Test Set')
plt.figure(figsize=(15,10))
plt.xlabel('Free Energy')
plt.ylabel('Probabilty')

plt.hist(costs[y_test == 0], bins = 100, color='green',
normed=1.0, label='Normal')

plt.hist(costs[y_test == 1], bins = 100, color='red', normed=1.0,
label = 'Anomaly')

plt.legend(loc="upper right")
plt.show()
```

Figure 5-66. Code to plot the free energy vs. the probability for each entry in the test set. All of the anomalies have free energies under 1500, so you can filter out all values for cost under 1500 to make the graph easier to visualize

The output should look somewhat like Figure 5-67.

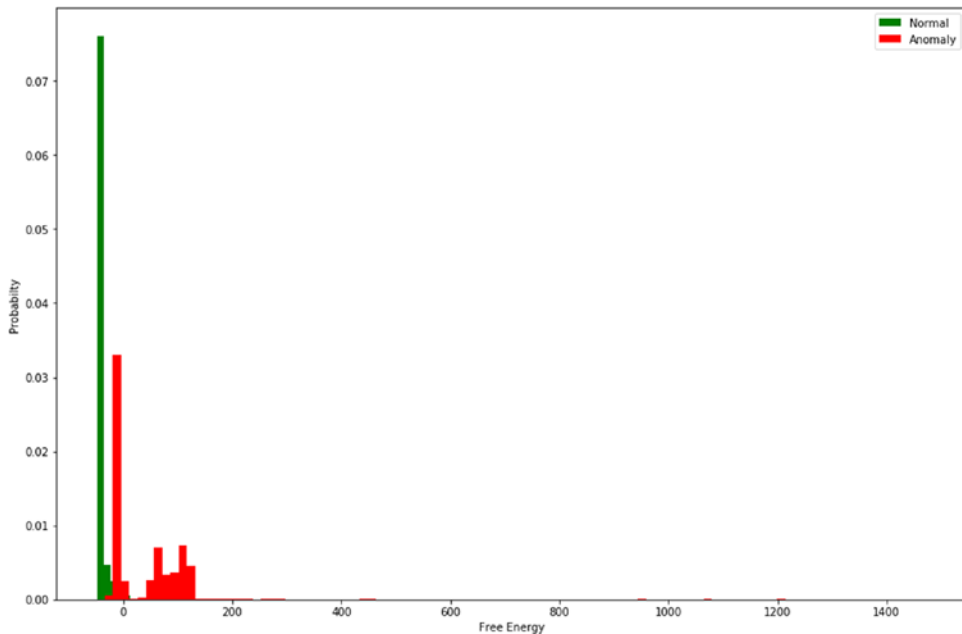


Figure 5-67. There seems to be a defined separation between the anomalies and the normal data points. The anomalies in general seem to have a much higher free energy cost and a lower-than-usual probability of occurring

Once again, the RBM has learned the distribution well enough that there's a clear and defined separation between the anomalies and the normal data entries.

Summary

In this chapter, we discussed restricted Boltzmann machines and how they can be used for anomaly detection. We also explored the application of the RBM to two data sets that represented two cases where standardization of the data is necessary for proper training. You now know more about what an RBM is, how it works, and how to apply it to different data sets.

In the next chapter, we will take a look at anomaly detection using recurrent neural networks.

CHAPTER 6

Long Short-Term Memory Models

In this chapter, you will learn about recurrent neural networks and long short-term memory models. You will also learn how LSTMs work and how they can be used to detect anomalies and how you can implement anomaly detection using LSTM. You will work through several datasets depicting time series of different types of data such as CPU utilization, taxi demand, etc. to illustrate how to detect anomalies. This chapter introduces you to many concepts using LSTM so as to enable you to explore further using the Jupyter notebooks provided as part of the book material.

In a nutshell, the following topics will be covered throughout this chapter:

- Sequences and time series analysis
- What is a RNN?
- What is an LSTM?
- LSTM applications

Sequences and Time Series Analysis

A time series is a series of data points indexed in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time. Thus, it is a sequence of discrete-time data. Examples of time series are ECG data, weather sensors, and stock prices.

Figure 6-1 shows some examples of time series.

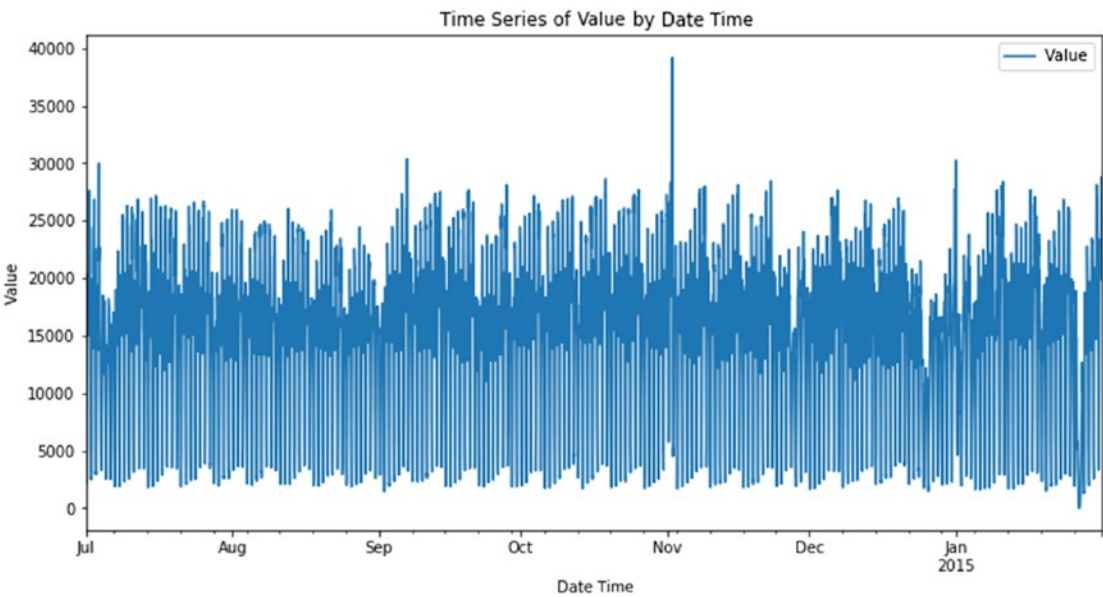


Figure 6-1. A time series

Figure 6-2 shows the monthly values of AMO index for last 150 years.

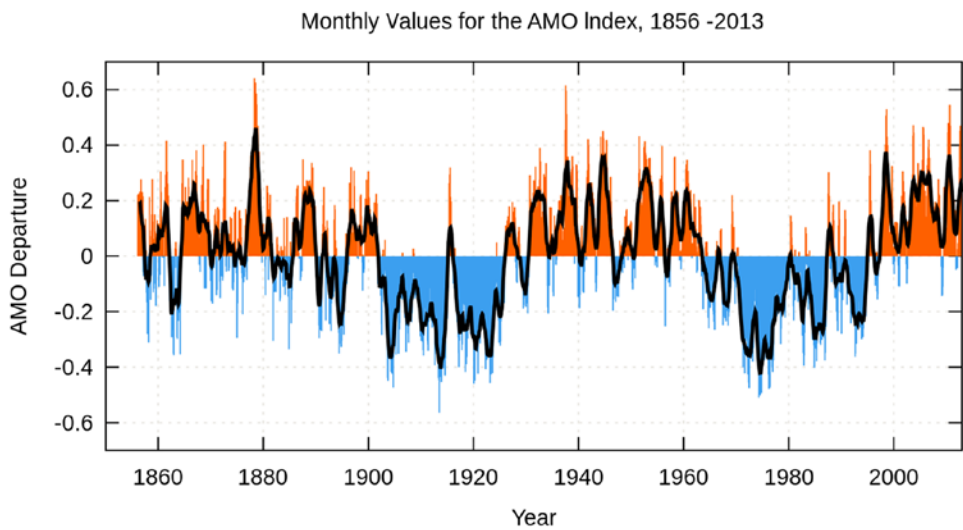


Figure 6-2. Monthly values of the AMO index

Figure 6-3 shows a chart of the BP stock price for a 20-year time period.

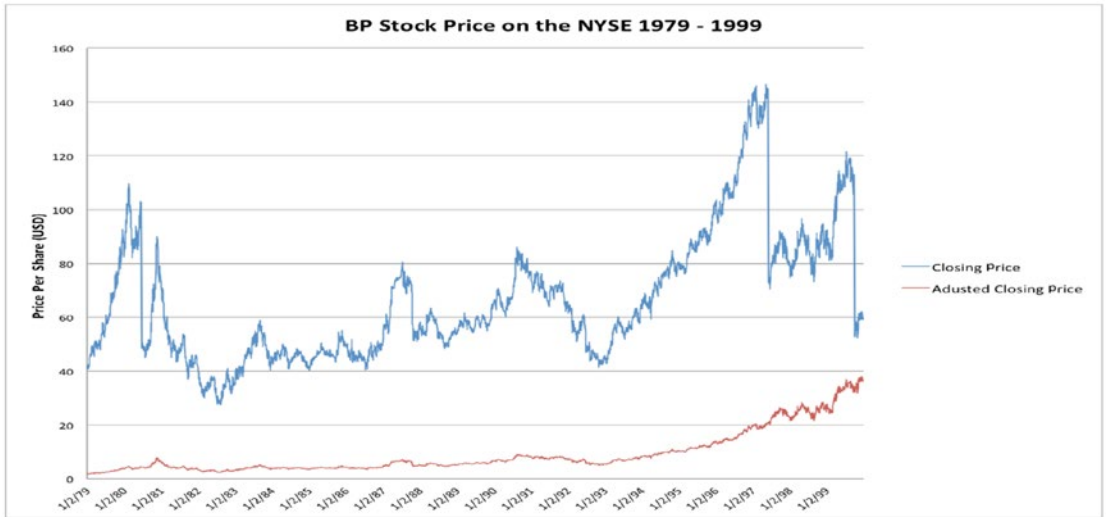


Figure 6-3. *BP stock price*

Time series analysis refers to the analysis of change in trends of data over a period of time. Time series analysis comprises methods for analyzing time series data in order to extract meaningful statistics and other characteristics of the data and has a variety of applications. One such application is the prediction of the future value of an item based on its past values. Future stock price prediction is probably the best example of such an application. Another very important use case is the ability to detect anomalies. By analyzing and learning the time series in terms of being able to understand the trends and changes seen from historical data, we can detect abnormal or anomalous data points in the time series.

Figure 6-4 is a time series with anomalies. It shows the normal data in green and possible anomalies in red.

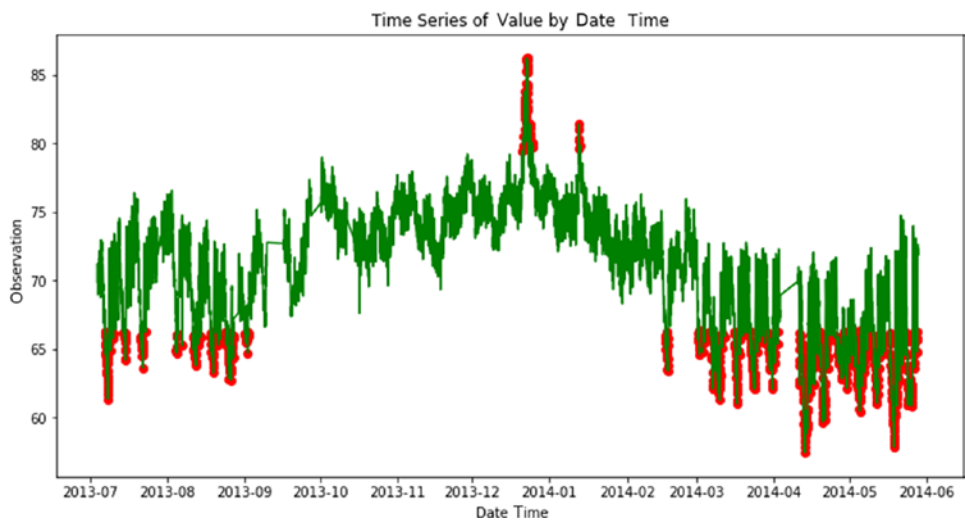


Figure 6-4. Time series with anomalies

What Is a RNN?

You have seen several types of neural networks throughout the book so you know that the high-level representation of neural networks looks like Figure 6-5.



Figure 6-5. A high-level representation of neural networks

Clearly, the neural network processes input and produces output, and this works on many types of input data with varying features. However, a critical piece to notice is that this neural network has no notion of the time of the occurrence of the event (input), only that input has come in.

So what happens with events (input) that come in as a stream over long periods of time? How can the neural network shown above handle trending in events, seasonality in events, etc.? How can it learn from the past and apply it to the present and future?

Recurrent neural networks try to address this by incrementally building neural networks, taking in signals from a previous timestamp into the current network. Figure 6-6 shows a RNN.

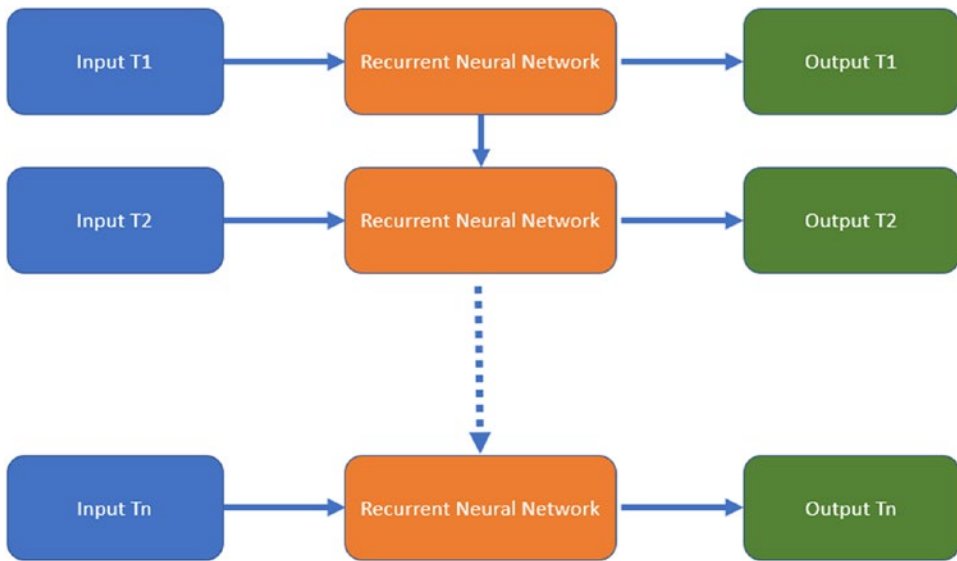


Figure 6-6. *A recurrent neural network*

You can see that RNN is a neural network with multiple layers or steps or stages. Each stage represents a time T ; the RNN at $T+1$ will consider the RNN at time T as one of the signals. Each stage passes its output to the next stage. The hidden state, which is passed from one stage to next, is the key for the RNN to work so well and this hidden state is analogous to some sort of memory retention. A RNN layer (or stage) acts as an encoder as it processes the input sequence and returns its own internal state. This state serves as the input of the decoder in the next stage, which is trained to predict the next point of the target sequence, given previous points of the target sequence. Specifically, it is trained to turn the target sequences into the same sequences but offset by one timestep in the future.

Backpropagation is used when training a RNN as in other neural networks, but in RNNs there is also a time dimension. In backpropagation, we take the derivative (gradient) of the loss with respect to each of the parameters. Using this information (loss), we can then shift the parameters in the opposite direction with a goal to minimize the loss. We have a loss at each timestep since we are moving through time and we can sum the losses across time to get the loss at each timestep. This is the same as summation of gradients across time.

The problem with the above recurrent neural networks, constructed from regular neural network nodes, is that as we try to model dependencies between sequence values that are separated by a significant number of other values, the gradients of timestep

T depends on gradients at $T-1$, gradients at $T-2$, and so on. This leads to the earliest gradient's contribution getting smaller and smaller as we move along the timesteps where the chain of gradients gets longer and longer. This is what is known as the vanishing gradient problem. This means the gradients of those earlier layers will become smaller and smaller and therefore the network won't learn long-term dependencies. RNN becomes biased as a result, only dealing with short-term data points.

LSTM networks are a way of solving this problem with RNNs.

What Is an LSTM?

A LSTM network is a kind of recurrent neural network. As seen above, a recurrent neural network is a neural network that attempts to model time or sequence dependent behavior, such as language, stock prices, weather sensors, and so on. This is performed by feeding back the output of a neural network layer at time T to the input of the same network layer at time $T + 1$. LSTM builds on top of the RNN, adding a memory component meant to help propagate the information learned at a time T to the future $T+1$, $T+2$, and so on. The main idea is that LSTM can forget irrelevant parts of previous state while selectively updating state and then outputting certain parts of the state that are relevant to the future.

How does this solve the vanishing gradient problem in RNNs? Well, now we are throwing some state, updating some state, and propagating forward some part of the state so we no longer have a long chain of backpropagation seen in RNNs. Thus, LSTMs are much more efficient than typical RNN.

Figure 6-7 is a RNN with tanh activation.

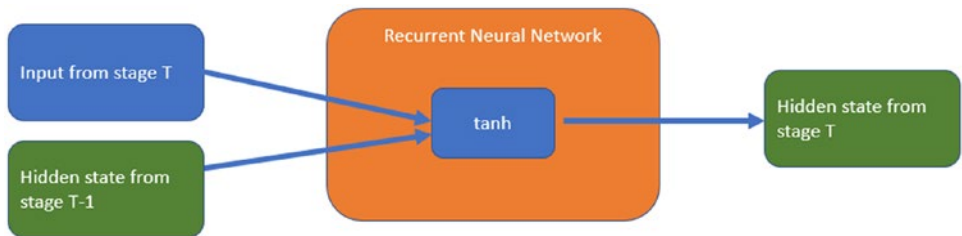


Figure 6-7. *A RNN with tanh activation*

The tanh function is called an activation function. There are several types of activation functions that help in applying non-linear transformations on the inputs at every node in the neural network. Figure 6-8 shows common activation functions.

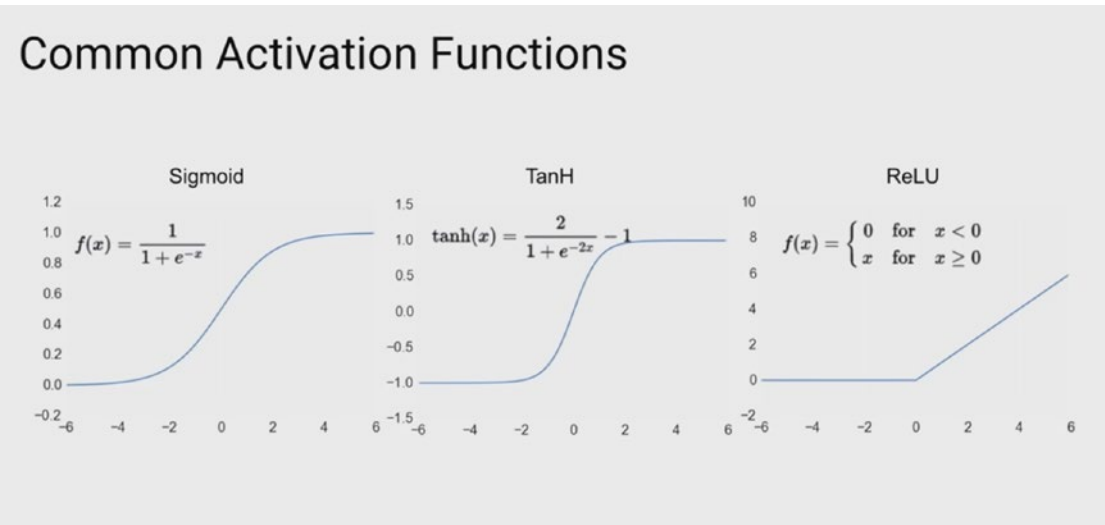


Figure 6-8. *Common activation functions*

The key idea behind activation functions is to add non-linearity to the data to align better with real-world problems and real-world data. In Figure 6-9, the top graph shows linearity and the bottom graph shows nonlinearity.

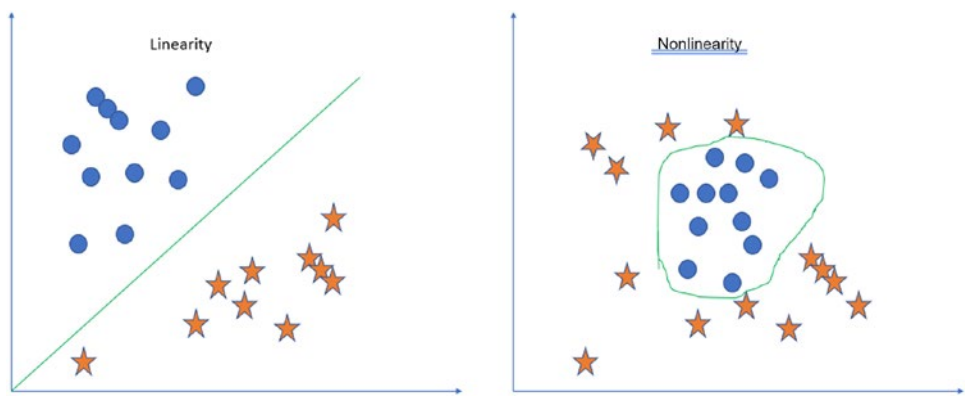


Figure 6-9. Linear and nonlinear data plots

Clearly, there is no linear equation to handle the nonlinearity so we need an activation function to deal with this property. The different activation functions are listed at <https://keras.io/activations/>.

In time series data, the data is spread over a period of time, not some instantaneous set such as seen in Chapter 4 autoencoders, for example. So not only it is important to look at the instantaneous data at some time T , it is also important for older historical data to the left of this point to be propagated through the steps in time. Since we need the signals from historical data points to survive for a long period of time, we need an activation function that can sustain information for a longer range before going to zero. $\tanh$ is the ideal activation function for the purpose and is graphed as shown in Figure 6-10.

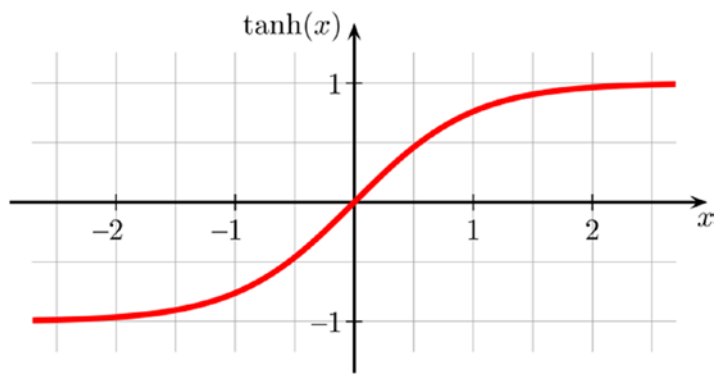


Figure 6-10. $\tanh$ activation

We also need sigmoid (another activation function) as a way to either remember or forget the information. A sigmoid activation function is shown in Figure 6-11.

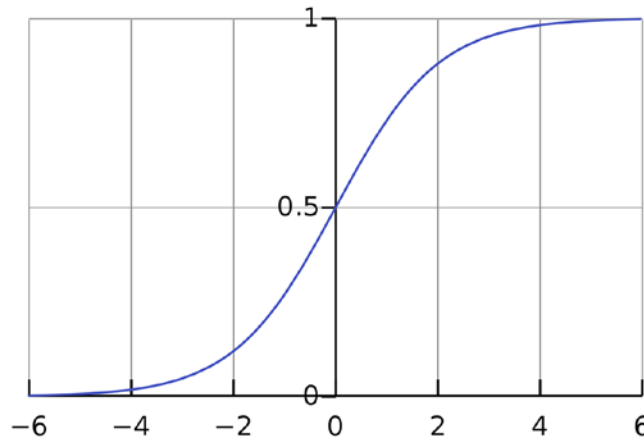


Figure 6-11. A sigmoid activation function

Now, conventional RNNs have a tendency to remember everything including unnecessary inputs which results in an inability to learn from long sequences. By contrast, LSTMs selectively remember important inputs and this allows them to handle both short-term and long-term dependencies.

So how does LSTM do this? It does this by releasing information between the hidden state and the cell state using three important gates: the forget gate, the input gate, and the output gate. A common LSTM unit is composed of a cell, an input gate, an output gate, and a forget gate. The cell remembers values over arbitrary time intervals and the three *gates* regulate the flow of information into and out of the cell.

A more detailed LSTM architecture is shown in Figure 6-12. There are a couple of key functions used, the tanh and the sigmoid, which are activation functions. F_t is the forget gate, I_t is the input gate, and O_t is the output gate.

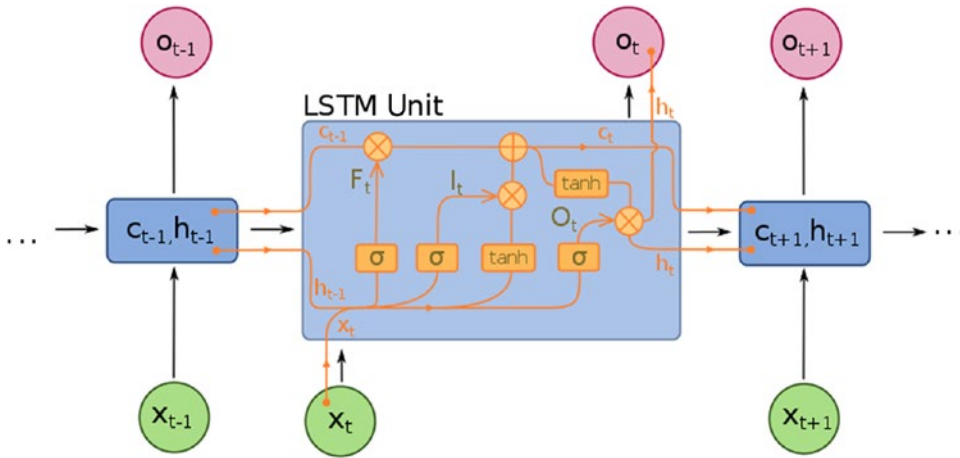


Figure 6-12. A detailed LSTM network
 Source: commons.wikimedia.org

A forget gate is the first part of the LSTM stage and pretty much decides how much information from a prior stage should be remembered or forgotten. This is accomplished by passing the previous hidden state h_{t-1} and current input x_t through a sigmoid function.

The input gate helps decide how much information to pass to current stage by using the sigmoid function and also a tanh function.

The output gate controls how much information will be retained by the hidden state of this stage and passed onto the next stage. Again, the current state passes through the tanh function.

Just for information, the compact forms of the equations for the forward pass of an LSTM unit with a forget gate are (source : Wikipedia)

$$\begin{aligned} f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\ i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\ o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\ c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\ h_t &= o_t \circ \sigma_h(c_t) \end{aligned}$$

where the initial values are $c_0 = 0$ and $h_0 = 0$, and the operator $\circ$ denotes the element-wise product. The subscript indexes the time step.

Variables

- $x_t \in \mathbb{R}^d$: Input vector to the LSTM unit
- $f_t \in \mathbb{R}^h$: Forget gate's activation vector
- $i_t \in \mathbb{R}^h$: Input/update gate's activation vector
- $o_t \in \mathbb{R}^h$: Output gate's activation vector
- $h_t \in \mathbb{R}^h$: Hidden state vector, also known as the output vector of the LSTM unit
- $c_t \in \mathbb{R}^h$: Cell state vector
- $W \in \mathbb{R}^{h \times d}$, $U \in \mathbb{R}^{h \times h}$ and $b \in \mathbb{R}^h$: Weight matrices and bias vector parameters, which need to be learned during training

The superscripts refer to the number of input features and number of hidden units, respectively.

σ_g : sigmoid function

σ_c : hyperbolic tangent function

σ_h : hyperbolic tangent function

LSTM for Anomaly Detection

In this section, you will look at LSTM implementations for some use cases using time series data as examples. You have few different time series datasets to use to try to detect anomalies using LSTM. All of them have a timestamp and a value that can easily be plotted in Python.

Figure 6-13 shows the basic code to import all necessary packages. Also note the versions of the various necessary packages.

```
import keras
from keras import optimizers
from keras import losses
from keras.models import Sequential, Model
from keras.layers import Dense, Input, Dropout, Embedding, LSTM
from keras.optimizers import RMSprop, Adam, Nadam
from keras.preprocessing import sequence
from keras.callbacks import TensorBoard

import sklearn
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, roc_auc_score
from sklearn.preprocessing import MinMaxScaler

import seaborn as sns
import pandas as pd
import numpy as np
import matplotlib

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
%matplotlib inline

import tensorflow
import sys
print("Python: ", sys.version)

print("pandas: ", pd.__version__)
print("numpy: ", np.__version__)
print("seaborn: ", sns.__version__)
print("matplotlib: ", matplotlib.__version__)
print("sklearn: ", sklearn.__version__)
print("Keras: ", keras.__version__)
print("Tensorflow: ", tensorflow.__version__)

Using TensorFlow backend.

Python: 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)]
pandas: 0.24.2
numpy: 1.16.3
seaborn: 0.9.0
matplotlib: 3.0.3
sklearn: 0.20.3
Keras: 2.2.4
Tensorflow: 1.13.1
```

Figure 6-13. Code to import packages

Figure 6-14 shows the code to visualize the results via a chart for the anomalies and a chart for the errors (the difference between predicted and truth) while training.


```

class Visualization:
    labels = ["Normal", "Anomaly"]

    def draw_anomaly(self, y, error, threshold):
        groupsDF = pd.DataFrame({'error': error,
                                'true': y}).groupby('true')

        figure, axes = plt.subplots(figsize=(12, 8))

        for name, group in groupsDF:
            axes.plot(group.index, group.error, marker='x' if name == 1 else 'o', linestyle='',
                      color='r' if name == 1 else 'g', label="Anomaly" if name == 1 else "Normal")

        axes.hlines(threshold, axes.get_xlim()[0], axes.get_xlim()[1], colors="b", zorder=100, label='')
        axes.legend()

        plt.title("Anomalies")
        plt.ylabel("Error")
        plt.xlabel("Data")
        plt.show()

    def draw_error(self, error, threshold):
        plt.figure(figsize=(10, 8))
        plt.plot(error, marker='o', ms=3.5, linestyle='',
                 label='Point')

        plt.hlines(threshold, xmin=0, xmax=len(error)-1, colors="r", zorder=100, label='Threshold')
        plt.legend()
        plt.title("Reconstruction error")
        plt.ylabel("Error")
        plt.xlabel("Data")
        plt.show()

```

Figure 6-14. Code to visualize errors and anomalies

You will use different examples of time series data to detect whether a point is normal/expected or abnormal/anomaly. Figure 6-15 shows the data being loaded into a Pandas dataframe. It shows a list of paths to datasets.

```

dataFilePaths = ['data/art_daily_no_noise.csv',
                  'data/art_daily_nojump.csv',
                  'data/art_daily_jumpsdown.csv',
                  'data/art_daily_perfect_square_wave.csv',
                  'data/art_increase_spike_density.csv',
                  'data/art_load_balancer_spikes.csv',
                  'data/ambient_temperature_system_failure.csv',
                  'data/nyc_taxi.csv',
                  'data/ec2_cpu_utilization.csv',
                  'data/rds_cpu_utilization.csv']

```

Figure 6-15. A list of paths to datasets

You will work with one of the datasets in more detail now. The dataset is `nyc_taxi`, which basically consists of timestamps and demand for taxis. This dataset shows the NYC taxi demand from 2014-07-01 to 2015-01-31 with an observation every half hour. There are few detectable anomalies in this dataset: Thanksgiving, Christmas, New Year's Day, a snow storm, etc.

Figure 6-16 shows the code to select the dataset.

```
i = 7

tensorlog = tensorlogs[i]
dataFilePath = dataFilePaths[i]
print("tensorlog: ", tensorlog)
print("dataFilePath: ", dataFilePath)

tensorlog: nyc_taxi
dataFilePath: data/nyc_taxi.csv
```

Figure 6-16. Code to select the dataset

You can load the data from the `dataFilePath` as a csv file using Pandas. Figure 6-17 shows the code to read the csv datafile into Pandas.

```
df = pd.read_csv(filepath_or_buffer=dataFilePath, header=0, sep=',')
print('Shape:', df.shape[0])
print('Head:')
print(df.head(5))

Shape: 10320
Head:
   timestamp  value
0  2014-07-01 00:00:00  10844
1  2014-07-01 00:30:00   8127
2  2014-07-01 01:00:00   6210
3  2014-07-01 01:30:00   4656
4  2014-07-01 02:00:00   3820
```

Figure 6-17. Code to read a csv datafile into Pandas

Figure 6-18 shows the plotting of the time series showing the months on the x-axis and the value on the y-axis. It also shows the code to generate a graph showing the time series.

```
df['Datetime'] = pd.to_datetime(df['timestamp'])
print(df.head(3))
df.shape
df.plot(x='Datetime', y='value', figsize=(12,6))
plt.xlabel('Date time')
plt.ylabel('Value')
plt.title('Time Series of value by date time')
```

```
      timestamp  value      Datetime
0 2014-07-01 00:00:00  10844 2014-07-01 00:00:00
1 2014-07-01 00:30:00   8127 2014-07-01 00:30:00
2 2014-07-01 01:00:00   6210 2014-07-01 01:00:00
```

```
Text(0.5, 1.0, 'Time Series of value by date time')
```

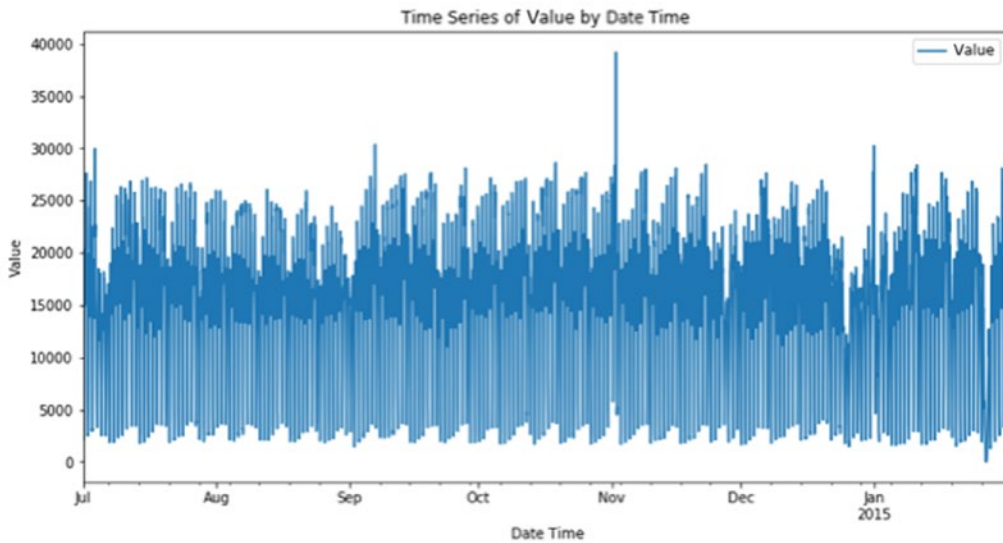


Figure 6-18. Plotting the time series

Let's understand the data more. You can run the `describe()` command to look at the value column. Figure 6-19 shows the code to describe the value column.

```
df.value.describe()
```

```
count    10320.000000
mean     15137.569380
std       6939.495808
min         8.000000
25%      10262.000000
50%      16778.000000
75%      19838.750000
max      39197.000000
Name: value, dtype: float64
```

Figure 6-19. Describing the value column

You can also plot the data using seaborn kde plot, as shown in Figure 6-20.

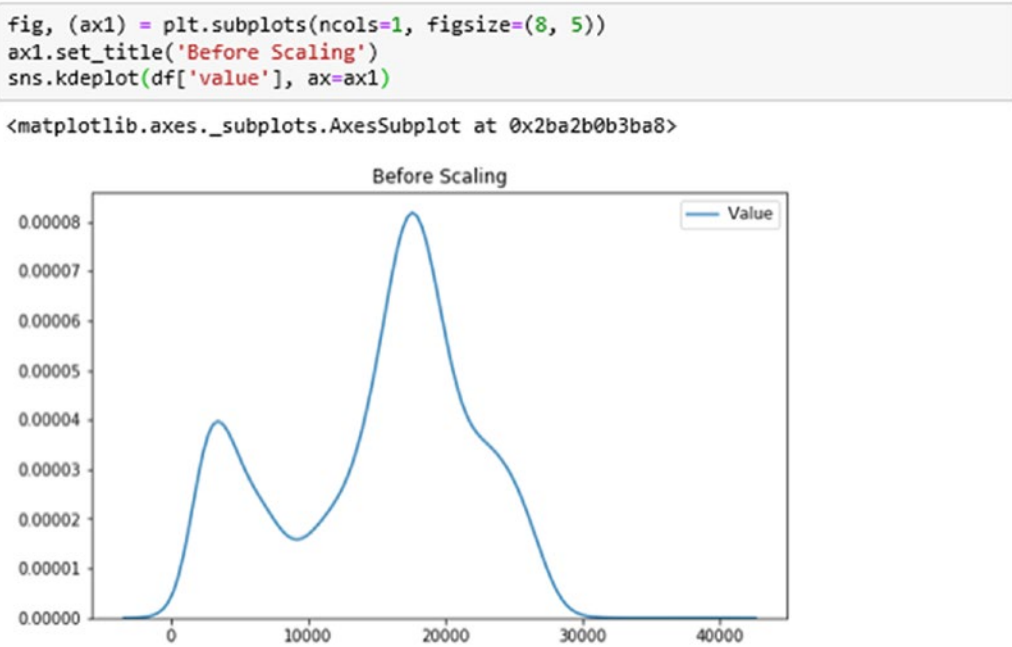


Figure 6-20. Using kde to plot the value column

The data points have a minimum of 8 and maximum of 39197, which is a wide range. You can use scaling to normalize the data.

The formula for scaling is $(x - \text{Min}) / (\text{Max} - \text{Min})$. Figure 6-21 shows the code to scale the data.

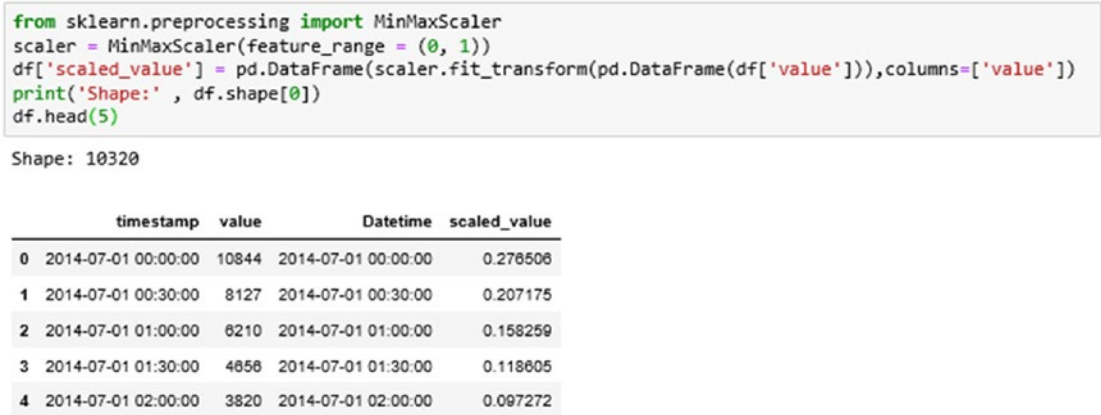


Figure 6-21. Code to scale the data

Now that you scaled the data, you can plot the data again. You can plot the data using seaborn kde plot, as shown in Figure 6-22.

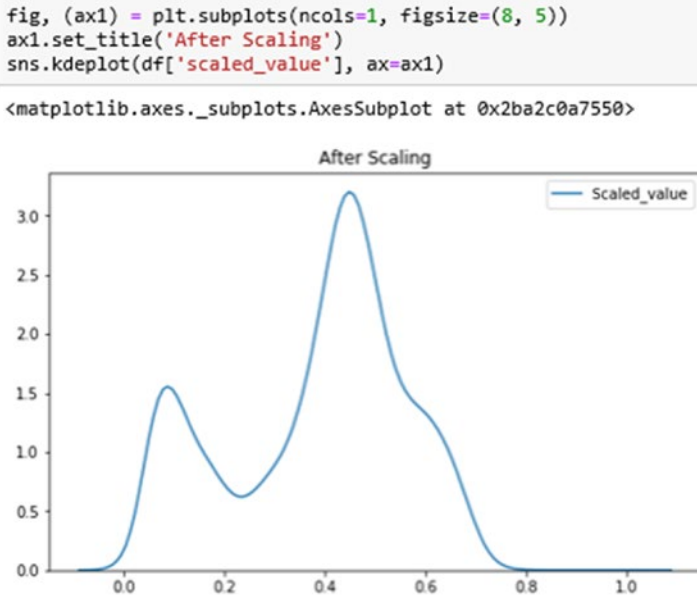


Figure 6-22. Using *kde* to plot the *scaled_value* column

You can take a look at the dataframe now that you have scaled the value column. Figure 6-23 shows the dataframe showing the timestamp and value as well as scaled_value and the datetime.

```
df.head(5)
```

	timestamp	value	Datetime	scaled_value
0	2014-07-01 00:00:00	10844	2014-07-01 00:00:00	0.276506
1	2014-07-01 00:30:00	8127	2014-07-01 00:30:00	0.207175
2	2014-07-01 01:00:00	6210	2014-07-01 01:00:00	0.158259
3	2014-07-01 01:30:00	4658	2014-07-01 01:30:00	0.118605
4	2014-07-01 02:00:00	3820	2014-07-01 02:00:00	0.097272

Figure 6-23. The modified dataframe

There are 10320 data points in the sequence and your goal is to find anomalies. This means you are trying to find out when data points are abnormal. If you can predict a data point at time T based on the historical data until T-1, then you have a way of looking at an expected value compared to an actual value to see if you are within the expected range of values for time T. If you predicted that ypred number of taxis are in demand on January 1, 2015, then you can compare this ypred with the actual yactual. The difference between ypred and yactual gives the error, and when you get the errors of all the points in the sequence, you end up with a distribution of just errors.

To accomplish this, you will use a sequential model using Keras. The model consists of a LSTM layer and a dense layer. The LSTM layer takes as input the time series data and learns how to learn the values with respect to time. The next layer is the dense layer (fully connected layer). The dense layer takes as input the output from the LSTM layer, and transforms it into a fully connected manner. Then, you apply a sigmoid activation on the dense layer so that the final output is between 0 and 1.

You also use the **adam** optimizer and the **mean squared error** as the loss function. Figure 6-24 shows the code to build a LSTM model.

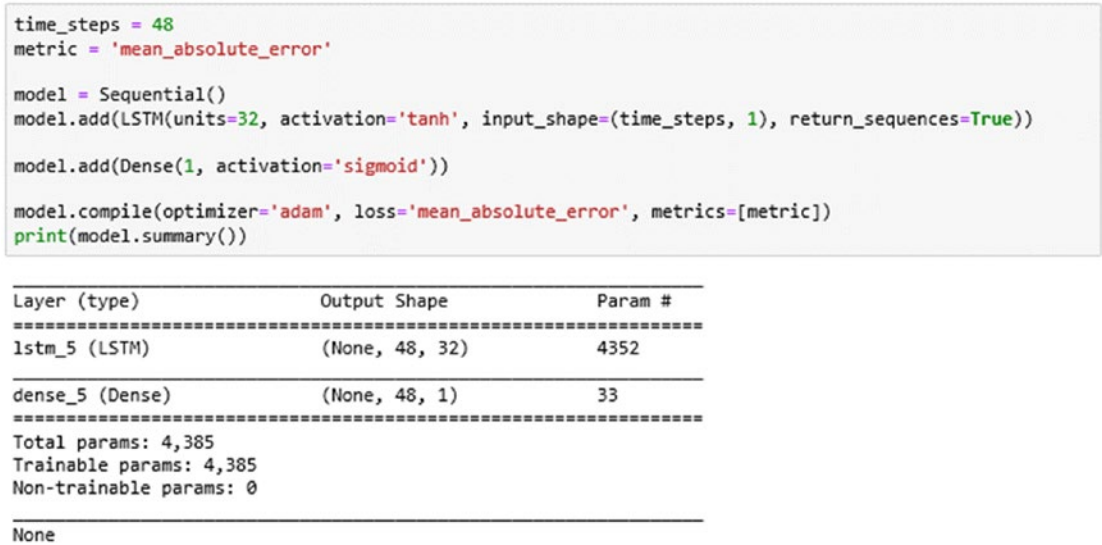


Figure 6-24. Code to build a LSTM model

As shown above, you used a LSTM layer. Let's look at the details of the LSTM layer function with all the possible parameters (Source: <https://keras.io/layers/recurrent/>):

```
keras.layers.LSTM(units, activation='tanh', recurrent_
activation='hard_sigmoid', use_bias=True, kernel_
initializer='glorot_uniform', recurrent_initializer='orthogonal',
bias_initializer='zeros', unit_forget_bias=True,
kernel_regularizer=None, recurrent_regularizer=None,
bias_regularizer=None, activity_regularizer=None,
kernel_constraint=None, recurrent_constraint=None,
bias_constraint=None, dropout=0.0, recurrent_dropout=0.0,
implementation=1, return_sequences=False, return_state=False,
go_backwards=False, stateful=False, unroll=False)
```

Arguments

- **units:** Positive integer, dimensionality of the output space
- **activation:** Activation function to use (see <https://keras.io/activations>). Default: hyperbolic tangent (tanh). If you pass None, no activation is applied (i.e. “linear” activation: $a(x) = x$).
- **recurrent_activation:** Activation function to use for the recurrent step (see <https://keras.io/activations>). Default: hard sigmoid (hard_sigmoid). If you pass None, no activation is applied (ie. “linear” activation: $a(x) = x$).
- **use_bias:** Boolean, whether the layer uses a bias vector
- **kernel_initializer:** Initializer for the kernel weights matrix, used for the linear transformation of the inputs (see <https://keras.io/initializers>)
- **recurrent_initializer:** Initializer for the recurrent_kernel weights matrix, used for the linear transformation of the recurrent state (see <https://keras.io/initializers>).
- **bias_initializer:** Initializer for the bias vector (see <https://keras.io/initializers>)

- **unit_forget_bias**: Boolean. If True, add 1 to the bias of the forget gate at initialization. Setting it to true will also force `bias_initializer="zeros"`. This is recommended in Jozefowicz et al. (2015).
- **kernel_regularizer**: Regularizer function applied to the kernel weights matrix (see <https://keras.io/regularizer>)
- **recurrent_regularizer**: Regularizer function applied to the recurrent_kernel weights matrix (see <https://keras.io/regularizer>)
- **bias_regularizer**: Regularizer function applied to the bias vector (see <https://keras.io/regularizer>)
- **activity_regularizer**: Regularizer function applied to the output of the layer (its “activation”) (see <https://keras.io/regularizer>)
- **kernel_constraint**: Constraint function applied to the kernel weights matrix (see <https://keras.io/constraints>)
- **recurrent_constraint**: Constraint function applied to the recurrent_kernel weights matrix (see <https://keras.io/constraints>)
- **bias_constraint**: Constraint function applied to the bias vector (see <https://keras.io/constraints>)
- **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
- **recurrent_dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.
- **implementation**: Implementation mode, either 1 or 2. Mode 1 will structure its operations as a larger number of smaller dot products and additions, whereas mode 2 will batch them into fewer, larger operations. These modes will have different performance profiles on different hardware and for different applications.
- **return_sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence.

- **return_state:** Boolean. Whether to return the last state in addition to the output. The returned elements of the state's list are the hidden state and the cell state, respectively.
- **go_backwards:** Boolean (default False). If True, process the input sequence backwards and return the reversed sequence.
- **stateful:** Boolean (default False). If True, the last state for each sample at index *i* in a batch will be used as the initial state for the sample of index *i* in the following batch.
- **unroll:** Boolean (default False). If True, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.

If you notice the LSTM call in the above code snippet, there is a parameter `time_steps=48` being used. This is the number of steps in the sequence that is used in training LSTM. 48 clearly means 24 hours, since your data points are 30 minutes apart. You can try changing this to 64 or 128 and see what happens to the output.

Figure 6-25 shows the code to split the sequence into a tumbling window of sub-sequences of length 48. Note the shape of `sequence_trimmed`, which is 215 subsequences of 48 points each with 1 dimension at each point (clearly you only have `scaled_value` as a column at each time stamp).

```
sequence = np.array(df['scaled_value'])
print(sequence)
time_steps = 48
samples = len(sequence)
trim = samples % time_steps
subsequences = int(samples/time_steps)
sequence_trimmed = sequence[:samples - trim]

print(samples, subsequences)
sequence_trimmed.shape = (subsequences, time_steps, 1)
print(sequence_trimmed.shape)

[0.27650616 0.20717548 0.1582587 ... 0.69664957 0.6783281 0.67059634]
10320 215
(215, 48, 1)
```

Figure 6-25. Code to create subsequences

Now, let's train your model for 20 epochs, using the training set as the validation data. You can do so as follows. Figure 6-26 shows the code to train the model.

```

training_dataset = sequence_trimmed
print("training_dataset: ", training_dataset.shape)

batch_size=32
epochs=20

model.fit(x=training_dataset, y=training_dataset,
          batch_size=batch_size, epochs=epochs,
          verbose=1, validation_data=(training_dataset, training_dataset),
          callbacks=[TensorBoard(log_dir='../logs/{0}'.format(tensorlog))])

training_dataset: (215, 48, 1)
Train on 215 samples, validate on 215 samples
Epoch 1/20
215/215 [=====] - 1s 6ms/step - loss: 0.0382 - mean_absolute_error: 0.0382 -
val_loss: 0.0377 - val_mean_absolute_error: 0.0377
Epoch 2/20
215/215 [=====] - 1s 5ms/step - loss: 0.0376 - mean_absolute_error: 0.0376 -
val_loss: 0.0370 - val_mean_absolute_error: 0.0370
Epoch 3/20
215/215 [=====] - 1s 5ms/step - loss: 0.0367 - mean_absolute_error: 0.0367 -
val_loss: 0.0361 - val_mean_absolute_error: 0.0361
Epoch 4/20
215/215 [=====] - 1s 5ms/step - loss: 0.0358 - mean_absolute_error: 0.0358 -
val_loss: 0.0354 - val_mean_absolute_error: 0.0354
Epoch 5/20
215/215 [=====] - 1s 5ms/step - loss: 0.0351 - mean_absolute_error: 0.0351 -
val_loss: 0.0346 - val_mean_absolute_error: 0.0346
Epoch 6/20
215/215 [=====] - 1s 5ms/step - loss: 0.0344 - mean_absolute_error: 0.0344 -
val_loss: 0.0339 - val_mean_absolute_error: 0.0339
Epoch 7/20
215/215 [=====] - 1s 5ms/step - loss: 0.0343 - mean_absolute_error: 0.0343 -
val_loss: 0.0332 - val_mean_absolute_error: 0.0332
Epoch 8/20
215/215 [=====] - 1s 5ms/step - loss: 0.0331 - mean_absolute_error: 0.0331 -
val_loss: 0.0326 - val_mean_absolute_error: 0.0326
Epoch 9/20
215/215 [=====] - 1s 5ms/step - loss: 0.0324 - mean_absolute_error: 0.0324 -
val_loss: 0.0319 - val_mean_absolute_error: 0.0319
Epoch 10/20
215/215 [=====] - 1s 5ms/step - loss: 0.0318 - mean_absolute_error: 0.0318 -
val_loss: 0.0315 - val_mean_absolute_error: 0.0315
Epoch 11/20
215/215 [=====] - 1s 5ms/step - loss: 0.0311 - mean_absolute_error: 0.0311 -
val_loss: 0.0309 - val_mean_absolute_error: 0.0309
Epoch 12/20
215/215 [=====] - 1s 5ms/step - loss: 0.0305 - mean_absolute_error: 0.0305 -
val_loss: 0.0299 - val_mean_absolute_error: 0.0299
Epoch 13/20
215/215 [=====] - 1s 5ms/step - loss: 0.0300 - mean_absolute_error: 0.0300 -
val_loss: 0.0302 - val_mean_absolute_error: 0.0302
Epoch 14/20
215/215 [=====] - 1s 5ms/step - loss: 0.0293 - mean_absolute_error: 0.0293 -
val_loss: 0.0289 - val_mean_absolute_error: 0.0289

```

Figure 6-26. Code to train the model

```

Epoch 15/20
215/215 [=====] - 1s 5ms/step - loss: 0.0286 - mean_absolute_error: 0.0286 -
val_loss: 0.0280 - val_mean_absolute_error: 0.0280
Epoch 16/20
215/215 [=====] - 1s 5ms/step - loss: 0.0278 - mean_absolute_error: 0.0278 -
val_loss: 0.0272 - val_mean_absolute_error: 0.0272
Epoch 17/20
215/215 [=====] - 1s 5ms/step - loss: 0.0270 - mean_absolute_error: 0.0270 -
val_loss: 0.0265 - val_mean_absolute_error: 0.0265
Epoch 18/20
215/215 [=====] - 1s 5ms/step - loss: 0.0265 - mean_absolute_error: 0.0265 -
val_loss: 0.0261 - val_mean_absolute_error: 0.0261
Epoch 19/20
215/215 [=====] - 1s 5ms/step - loss: 0.0260 - mean_absolute_error: 0.0260 -
val_loss: 0.0254 - val_mean_absolute_error: 0.0254
Epoch 20/20
215/215 [=====] - 1s 6ms/step - loss: 0.0251 - mean_absolute_error: 0.0251 -
val_loss: 0.0248 - val_mean_absolute_error: 0.0248

```

Figure 26. (continued)

Figure 6-27 shows the plotting of the loss during the training process through the epochs of training.



Figure 6-27. Graph of loss in TensorBoard

Figure 6-28 shows the plotting of the mean absolute error during the training process through the epochs of training.

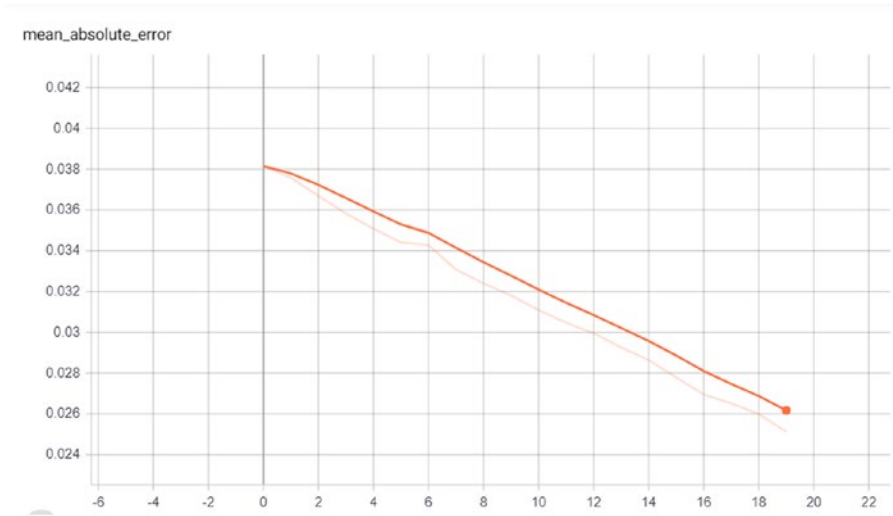


Figure 6-28. Graph of mean absolute error in TensorBoard

Figure 6-29 shows the plotting of the loss of validation during the training process through the epochs of training.

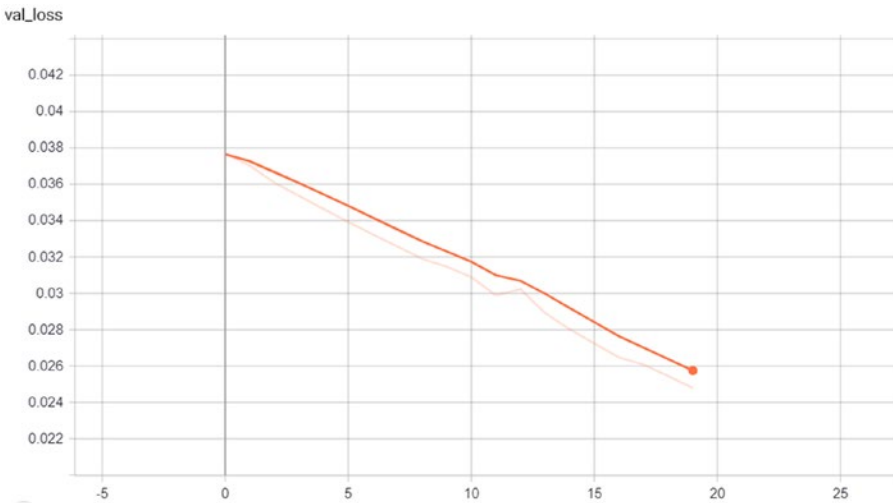


Figure 6-29. Graph of loss of validation in TensorBoard

Figure 6-30 shows the plotting of the mean absolute error of validation during the training process through the epochs of training.

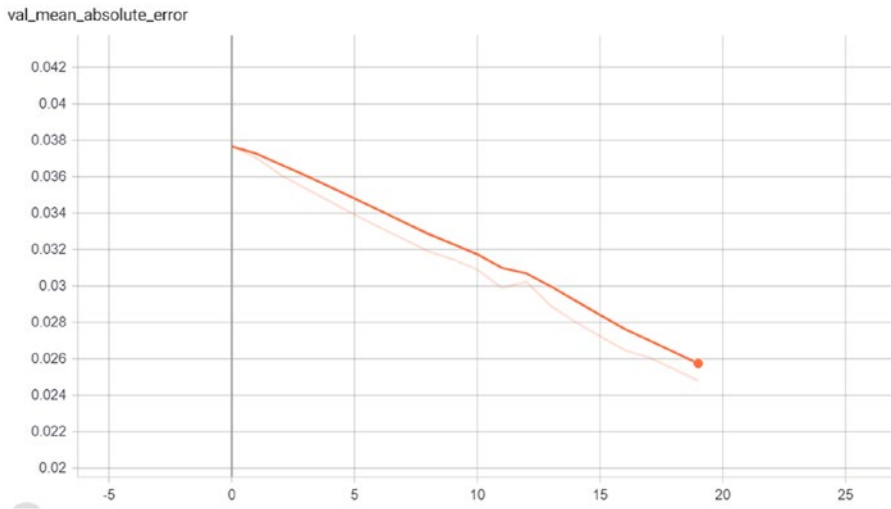


Figure 6-30. Graph of mean absolute error of validation in TensorBoard

Figure 6-31 shows the graph of the model as visualized by TensorBoard.

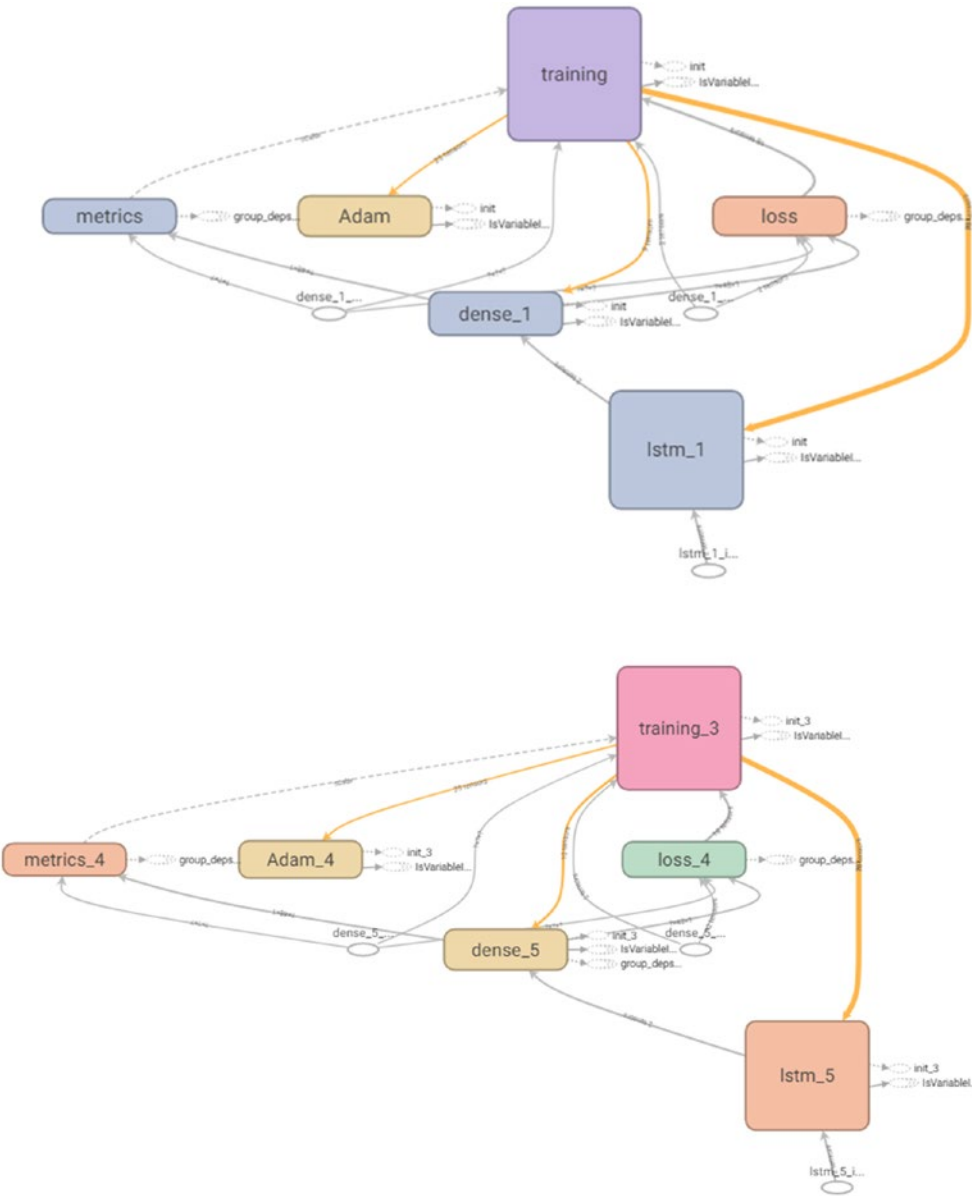


Figure 6-31. Graph of the model as visualized by TensorBoard

Once the model is trained, you can predict a test dataset that is split into subsequences of the same length (time_steps) as the training datasets. Once this is done, you can then compute the root mean square error (RMSE).

Figure 6-32 shows the code to predict on the testing dataset.

```
import math
from sklearn.metrics import mean_squared_error

sequence = np.array(df['scaled_value'])
print(sequence)
time_steps = 48
samples = len(sequence)
trim = samples % time_steps
subsequences = int(samples/time_steps)
sequence_trimmed = sequence[:samples - trim]

print(samples, subsequences)
sequence_trimmed.shape = (subsequences, time_steps, 1)
print(sequence_trimmed.shape)

testing_dataset = sequence_trimmed
print("testing_dataset: ", testing_dataset.shape)

testing_pred = model.predict(x=testing_dataset)
print("testing_pred: ", testing_pred.shape)

testing_dataset = testing_dataset.reshape((testing_dataset.shape[0]*testing_dataset.shape[1]), testing_dataset.shape[2])
print("testing_dataset: ", testing_dataset.shape)

testing_pred = testing_pred.reshape((testing_pred.shape[0]*testing_pred.shape[1]), testing_pred.shape[2])
print("testing_pred: ", testing_pred.shape)
errorsDF = testing_dataset - testing_pred
print(errorsDF.shape)
rmse = math.sqrt(mean_squared_error(testing_dataset, testing_pred))
print('Test RMSE: %.3f' % rmse)
```

< 0.27650616 0.20717548 0.1582587 ... 0.69664957 0.6783281 0.67059634] 10320 215 (215, 48, 1) testing_dataset: (215, 48, 1) testing_pred: (215, 48, 1) testing_dataset: (10320, 1) testing_pred: (10320, 1) (10320, 1) Test RMSE: 0.040

Figure 6-32. Code to predict on the testing dataset

RMSE is 0.040, which is quite low, and this is also evident from the low loss from the training phase after 20 epochs: **loss: 0.0251 - mean_absolute_error: 0.0251 - val_loss: 0.0248 - val_mean_absolute_error: 0.0248**

Now you can use the predicted dataset and the test dataset to compute the difference as diff, which is then passed through vector norms. Calculating the length or magnitude of vectors is often required directly as a regularization method in machine learning. Then you can sort the scores/diffs and use a cutoff value to pick the threshold. This obviously can change as per the parameters you choose, particularly the cutoff value (which is 0.99 in Figure 6-33). The figure also shows the code to compute the threshold.


```
#based on cutoff after sorting errors
dist = np.linalg.norm(testing_dataset - testing_pred, axis=-1)

scores = dist.copy()
print(scores.shape)
scores.sort()
cutoff = int(0.999 * len(scores))
print(cutoff)
#print(scores[cutoff:])
threshold = scores[cutoff]
print(threshold)

(10320,)
10309
0.3330642728290365
```

Figure 6-33. Code to compute the threshold

You got 0.333 as the threshold; anything above is considered an anomaly.

Figure 6-34 shows the code to plot testing dataset (GREEN) and the corresponding predicted dataset (RED).

```
plt.figure(figsize=(24,16))
plt.plot(testing_dataset, color='green')
plt.plot(testing_pred, color='red')

[<matplotlib.lines.Line2D at 0x2bc082169e8>]
```

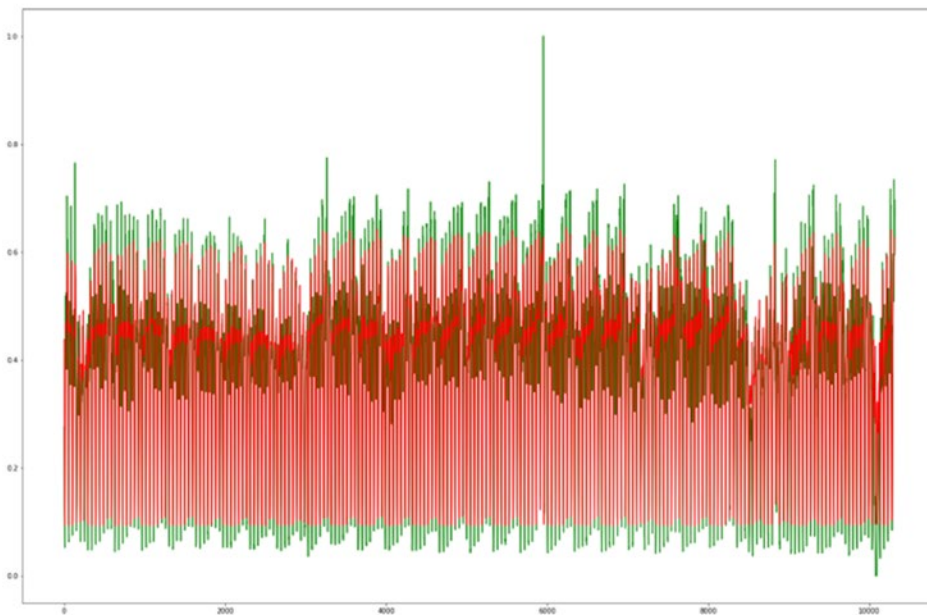


Figure 6-34. Plotting the testing and predicted datasets

Figure 6-35 shows the code to classify a datapoint as anomaly or normal.

```
#label the records anomalies or not based on threshold
z = zip(dist >= threshold, dist)

y_label=[]
error = []
for idx, (is_anomaly, dist) in enumerate(z):
    if is_anomaly:
        y_label.append(1)
    else:
        y_label.append(0)
    error.append(dist)
```

Figure 6-35. Code to classify a datapoint as anomaly or normal

Figure 6-36 shows the code to plot the data points with respect to the threshold.

```
viz = Visualization()
viz.draw_anomaly(y_label, error, threshold)
```

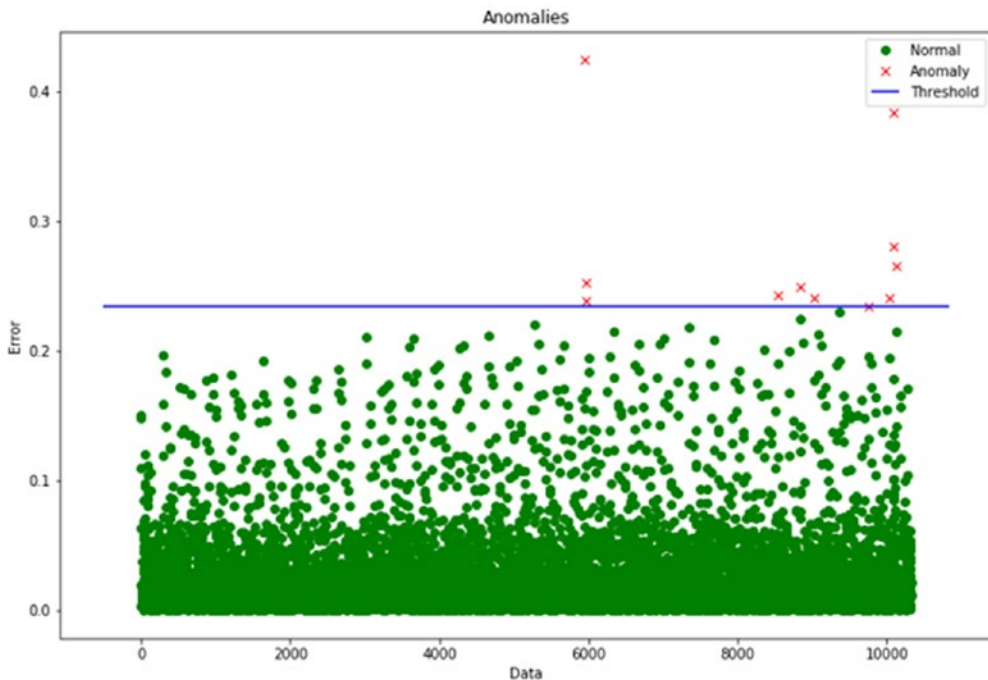


Figure 6-36. Code to plot the data points with respect to the threshold

Figure 6-37 shows the code to append the anomaly flag to the dataframe.

```
adf = pd.DataFrame({'Datetime': df['Datetime'], 'observation': df['value'],
                    'error': error, 'anomaly': y_label})
adf.head(5)
```

	Datetime	observation	error	anomaly
0	2014-07-01 00:00:00	10844	0.150302	0
1	2014-07-01 00:30:00	8127	0.147902	0
2	2014-07-01 01:00:00	6210	0.109466	0
3	2014-07-01 01:30:00	4656	0.063570	0
4	2014-07-01 02:00:00	3820	0.019833	0

Figure 6-37. Code to append the anomaly flag to the dataframe

Figure 6-38 shows the code to generate a graph showing the anomalies.

```
figure, axes = plt.subplots(figsize=(12, 6))
axes.plot(adf['Datetime'], adf['observation'], color='g')
anomaliesDF = adf.query('anomaly == 1')
axes.scatter(anomaliesDF['Datetime'].values, anomaliesDF['observation'], color='r')
plt.xlabel('Date time')
plt.ylabel('observation')
plt.title('Time Series of value by date time')
Text(0.5, 1.0, 'Time Series of value by date time')
```

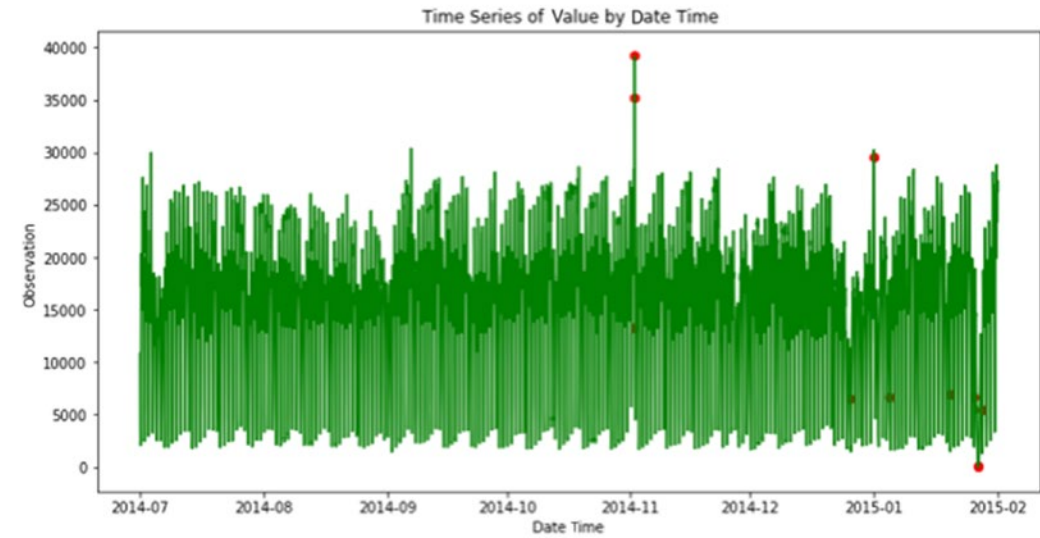


Figure 6-38. A graph showing anomalies

In above graph you can spot an anomaly around Thanksgiving Day, one around New Year Eve, and another one possibly on a snow storm day in January.

If you play around with some of the parameters you used, such as number of time_steps, threshold cutoffs, epochs of the neural network, batch size, and hidden layer, you will see different results.

A good way to improve the detection is to curate good normal data, use identified anomalies, and put it in the mix to have a way to tune the parameters until you get good matches on the identified anomalies.

Examples of Time Series

art_daily_no_noise

This data set has no noise or anomalies and is a normal time series dataset. As you can see below, the time series has values at different timestamps.

Dataset: art_daily_no_noise.csv

Figure 6-39 shows the code to generate a graph showing the time series.

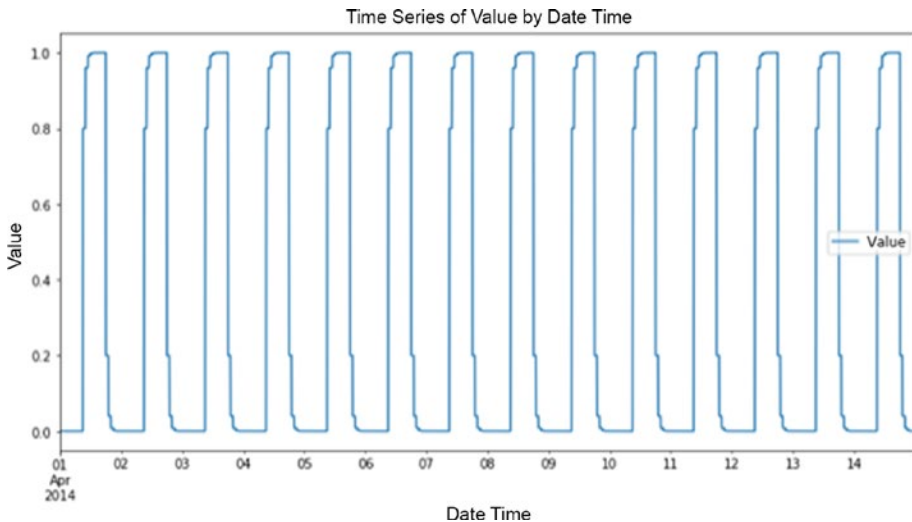


Figure 6-39. A graph showing the time series

Using visualization, you can plot the new time series now. As shown below, the time series shows the datetime vs. the value column. Since there are no anomalies, everything is green. Figure 6-40 shows code to generate a graph showing anomalies.

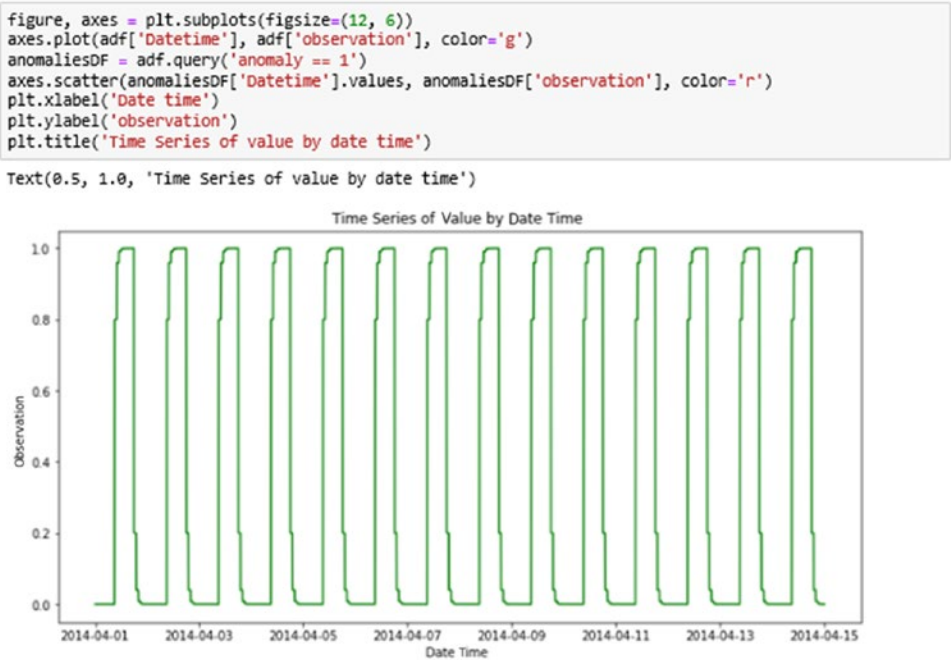


Figure 6-40. A graph showing anomalies

Since this data set has no noise or anomalies and is a normal time series dataset, there are no anomalies (datapoints in RED) shown and everything is green.

Next, let's examine another dataset which is different from the current dataset. You will build a LSTM model and see if there are anomalies or not.

art_daily_nojump

This data set has no noise or anomalies and is a normal time series dataset. As you can see below, the time series has values at different timestamps.

Using visualization, you can plot the time series now. You convert the timestamp to datetime for this work and also drop the timestamp column. As shown below, the time series shows the datetime vs. the value column.

Dataset: art_daily_nojump.csv

Figure 6-41 shows the code to generate a graph showing the time series.

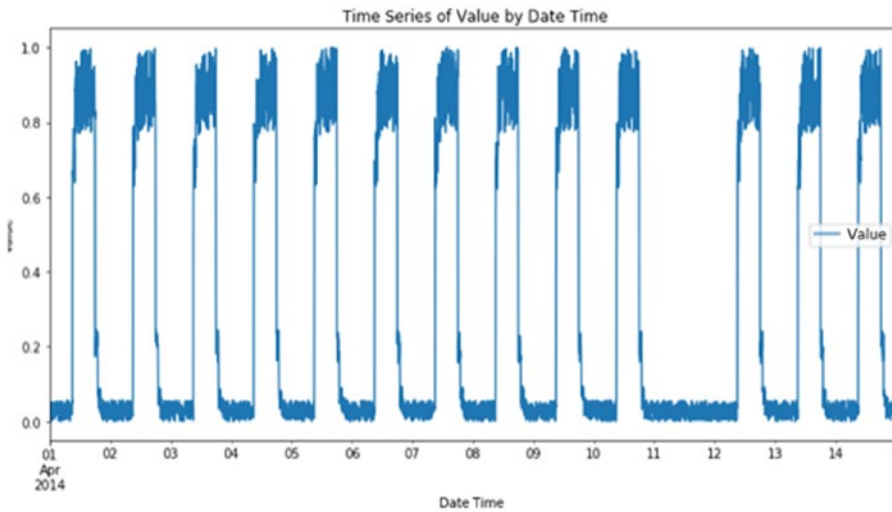


Figure 6-41. *A graph showing the time series*

Let's add the anomaly column to the original dataframe and prepare a new dataframe. Using visualization, you can plot the new time series now. As shown below, the time series shows the datetime vs. the value column. Since there are no anomalies, everything is green. Figure 6-42 shows the code to generate a graph showing anomalies.

```
figure, axes = plt.subplots(figsize=(12, 6))
axes.plot(adf['Datetime'], adf['observation'], color='g')
anomaliesDF = adf.query('anomaly == 1')
axes.scatter(anomaliesDF['Datetime'], anomaliesDF['observation'], color='r')
plt.xlabel('Date time')
plt.ylabel('observation')
plt.title('Time Series of value by date time')
```

```
Text(0.5, 1.0, 'Time Series of value by date time')
```

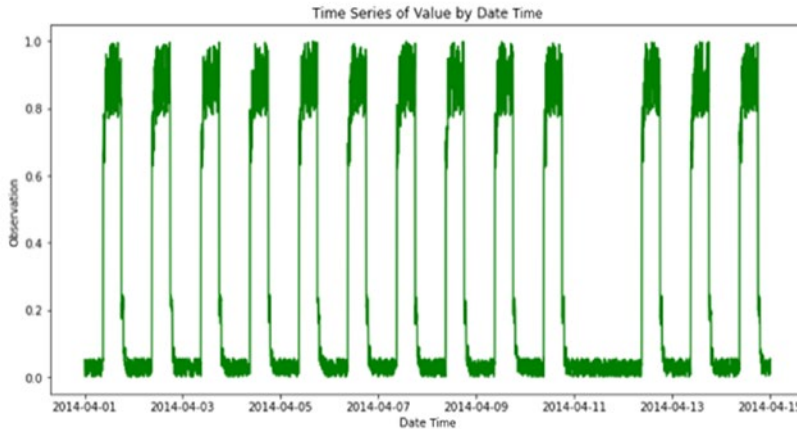


Figure 6-42. A graph showing anomalies

Since this data set has no noise or anomalies and is a normal time series dataset, there are no anomalies (datapoints in RED) shown and everything is green.

Next, let's examine another dataset which is different from the current dataset. You will build a LSTM model and see if there are anomalies or not.

art_daily_jumpsdown

This data set has mixture of normal data and anomalies. As you can see below, the time series has values at different timestamps.

Using visualization, you can plot the time series now. You convert the timestamp to datetime for this work and also drop the timestamp column. As shown below, the time series shows the datetime vs. the value column.

Dataset: art_daily_jumpsdown.csv

Figure 6-43 shows the code to generate a graph showing the time series.

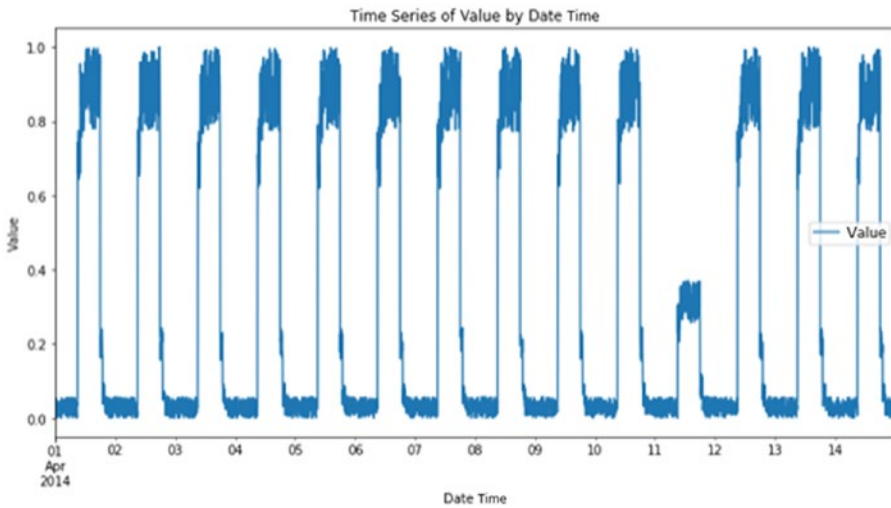


Figure 6-43. *A graph showing the time series*

Let's add the anomaly column to the original dataframe and prepare a new dataframe. Using visualization, you can plot the new time series now. As shown below, the time series shows the datetime vs. the value column. Normal data points are shown in green and anomalies are shown in red. Figure 6-44 shows the code to generate a graph showing anomalies.

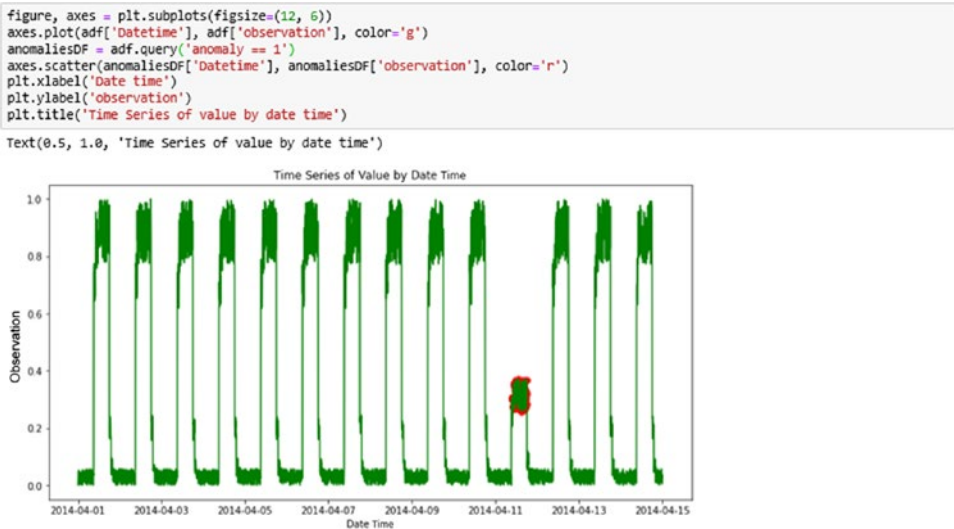


Figure 6-44. A graph showing anomalies

Since this data set has some noise or anomalies, there are anomalies (datapoints in RED) shown and everything else that is normal is green.

Next, let's examine another dataset which is different from the current dataset. You will build a LSTM model and see if there are anomalies or not.

art_daily_perfect_square_wave

This data set has no noise or anomalies and is a normal time series dataset. As you can see below, the time series has values at different timestamps.

Using visualization, you can plot the time series now. You convert the timestamp to datetime for this work and also drop the timestamp column. As shown below, the time series shows the datetime vs. the value column.

Dataset: art_daily_perfect_square_wave.csv

Figure 6-45 shows the code to generate a graph showing the time series.

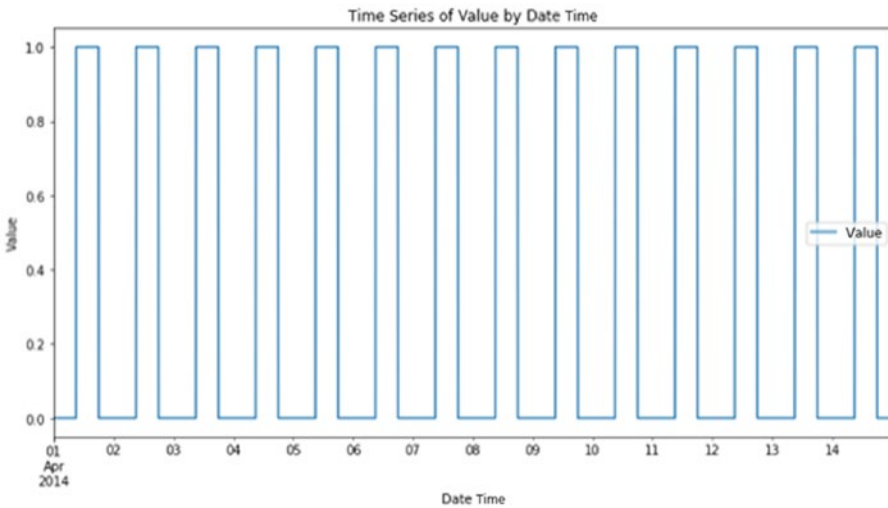


Figure 6-45. A graph showing the time series

Let's add the anomaly column to the original dataframe and prepare a new dataframe. Using visualization, you can plot the new time series now. As shown below, the time series shows the datetime vs. value column. Since there are no anomalies, everything is green. Figure 6-46 shows the code to generate a graph showing anomalies.

```
figure, axes = plt.subplots(figsize=(12, 6))
axes.plot(adf['Datetime'], adf['observation'], color='g')
anomaliesDF = adf.query('anomaly == 1')
axes.scatter(anomaliesDF['Datetime'], anomaliesDF['observation'], color='r')
plt.xlabel('Date time')
plt.ylabel('observation')
plt.title('Time Series of value by date time')
Text(0.5, 1.0, 'Time Series of value by date time')
```

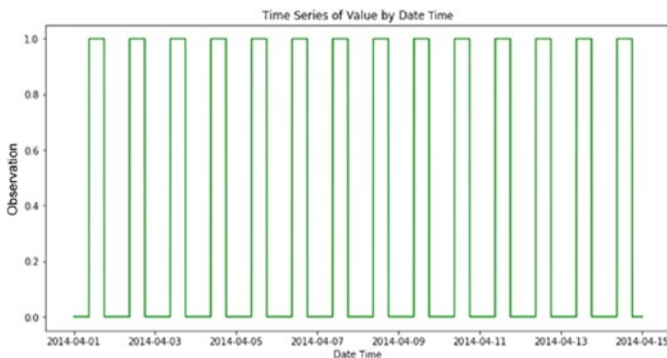


Figure 6-46. A graph showing anomalies

Since this data set has no noise or anomalies and is a normal time series dataset, there are no anomalies (datapoints in RED) shown and everything is green.

Next, let’s examine another dataset which is different from the current dataset. You will build a LSTM model and see if there are anomalies or not.

art_load_balancer_spikes

This data set has mixture of normal data and anomalies. As you can see below, the time series has values at different timestamps.

Using visualization, you can plot the time series now. You convert the timestamp to datetime for this work and also drop the timestamp column. As shown below, the time series shows the datetime vs. the value column.

Dataset: art_load_balancer_spikes.csv

Figure 6-47 shows the code to generate a graph showing the time series.

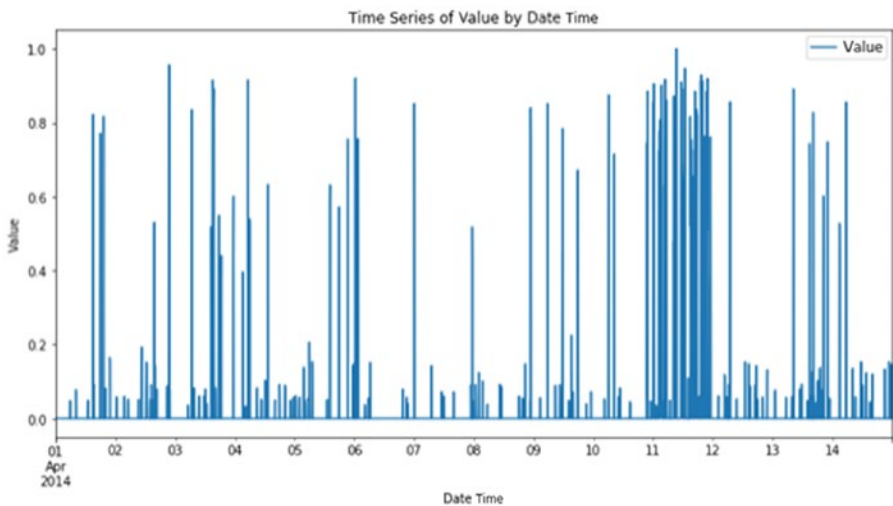


Figure 6-47. A graph showing the time series

Let’s add the anomaly column to the original dataframe and prepare a new dataframe. Using visualization, you can plot the new time series now. As shown below, the time series shows the datetime vs. the value column. Normal data points are shown in green and anomalies are shown in red. Figure 6-48 shows the code to generate a graph showing anomalies.

```
figure, axes = plt.subplots(figsize=(12, 6))
axes.plot(adf['Datetime'], adf['observation'], color='g')
anomaliesDF = adf.query('anomaly == 1')
axes.scatter(anomaliesDF['Datetime'], anomaliesDF['observation'], color='r')
plt.xlabel('Date time')
plt.ylabel('observation')
plt.title('Time Series of value by date time')
Text(0.5, 1.0, 'Time Series of value by date time')
```

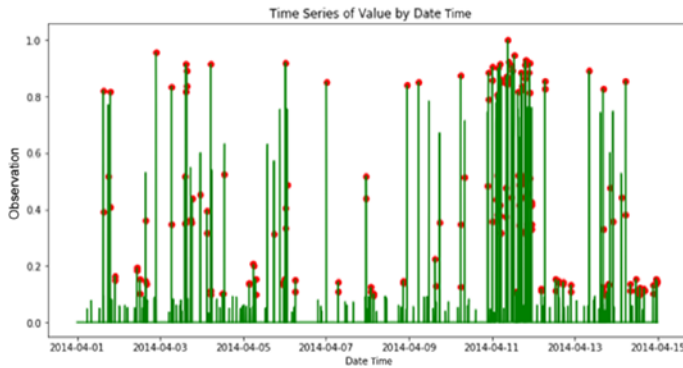


Figure 6-48. A graph showing anomalies

Since this data set has some noise or anomalies, there are anomalies (datapoints in RED) shown and everything else that is normal is green.

Next, let's examine another dataset which is different from the current dataset. You will build a LSTM model and see if there are anomalies or not.

ambient_temperature_system_failure

This data set has mixture of normal data and anomalies. As you can see below, the time series has values at different timestamps.

Using visualization, you can plot the time series now. You convert the timestamp to datetime for this work and also drop the timestamp column. As shown below, the time series shows the datetime vs. the value column.

Dataset: ambient_temperature_system_failure.csv

Figure 6-49 shows the code to generate a graph showing the time series.

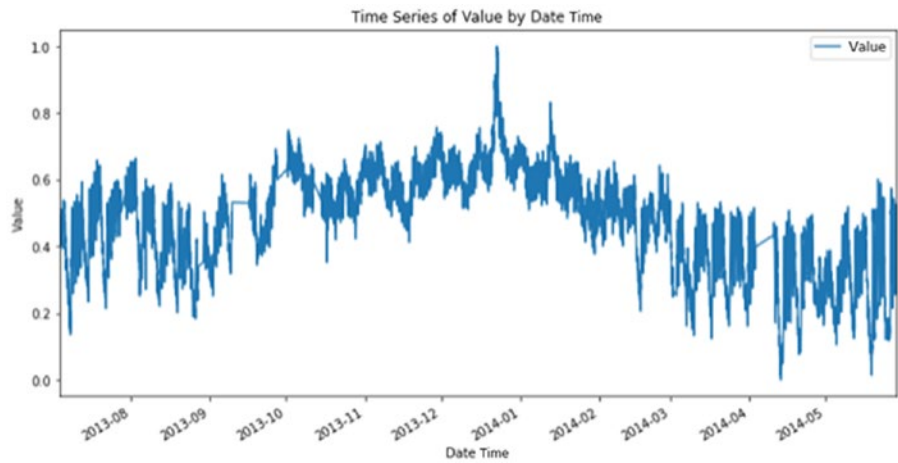


Figure 6-49. A graph showing the time series

Let’s add the anomaly column to the original dataframe and prepare a new dataframe. Using visualization, you can plot the new time series now. As shown below, the time series shows the datetime vs. the value column. Normal data points are shown in green and anomalies are shown in red. Figure 6-50 shows the code to generate a graph showing anomalies.

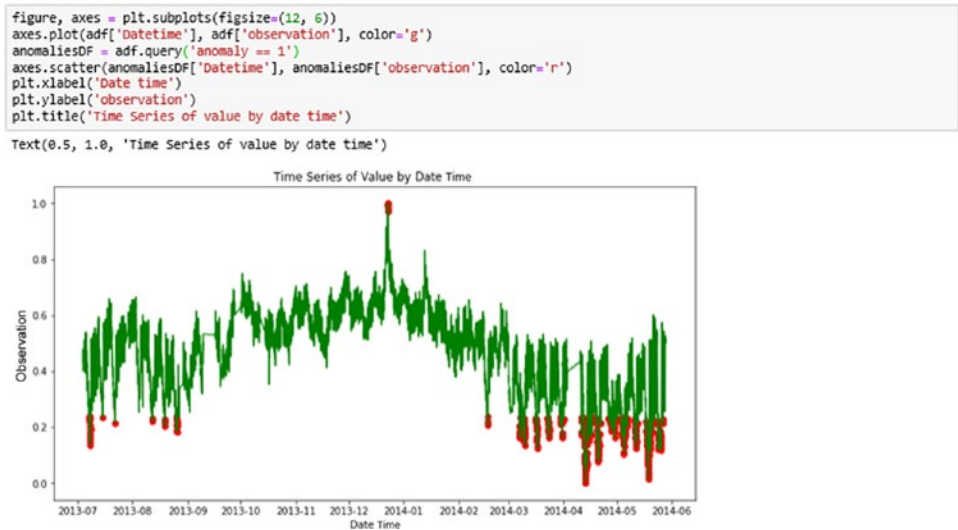


Figure 6-50. A graph showing anomalies

Since this data set has some noise or anomalies, there are anomalies (datapoints in RED) shown and everything else that is normal is green.

Next, let's examine another dataset which is different from the current dataset. You will build a LSTM model and see if there are anomalies or not.

ec2_cpu_utilization

This data set has mixture of normal data and anomalies. As you can see below, the time series has values at different timestamps.

Using visualization, you can plot the time series now. You convert the timestamp to datetime for this work and also drop the timestamp column. As shown below, the time series shows the datetime vs. the value column.

Dataset: ec2_cpu_utilization.csv

Figure 6-51 shows the code to generate a graph showing the time series.

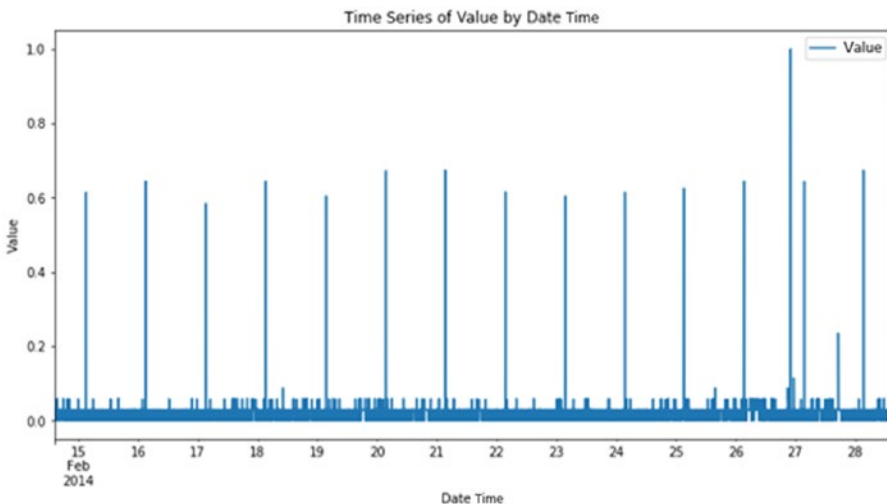


Figure 6-51. A graph showing the time series

Let's add the anomaly column to the original dataframe and prepare a new dataframe. Using visualization, you can plot the new time series now. As shown below, the time series shows the datetime vs. the value column. Normal data points are shown in green and anomalies are shown in red. Figure 6-52 shows the code to generate a graph showing anomalies.

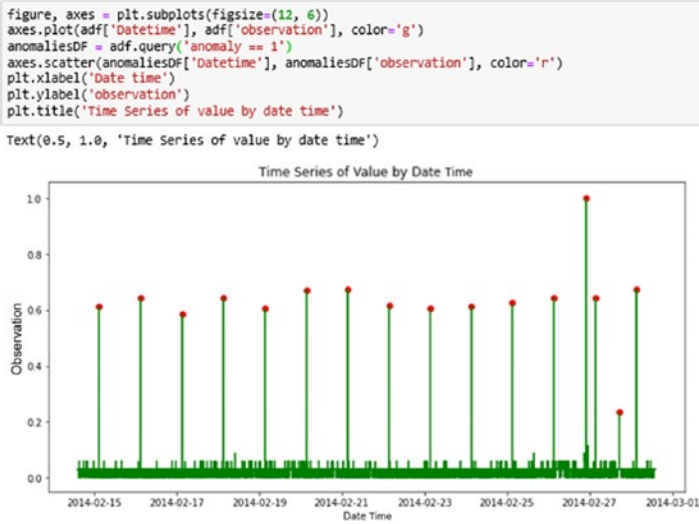


Figure 6-52. A graph showing anomalies

Since this data set has some noise or anomalies, there are anomalies (datapoints in RED) shown and everything else that is normal is green.

Next, let's examine another dataset which is different from the current dataset. You will build a LSTM model and see if there are anomalies or not.

rds_cpu_utilization

This data set has mixture of normal data and anomalies. As you can see below, the time series has values at different timestamps.

Using visualization, you can plot the time series now. You convert the timestamp to datetime for this work and also drop the timestamp column. As shown below, the time series shows the datetime vs. the value column.

Dataset: rds_cpu_utilization.csv

Figure 6-53 shows the code to generate a graph showing the time series.

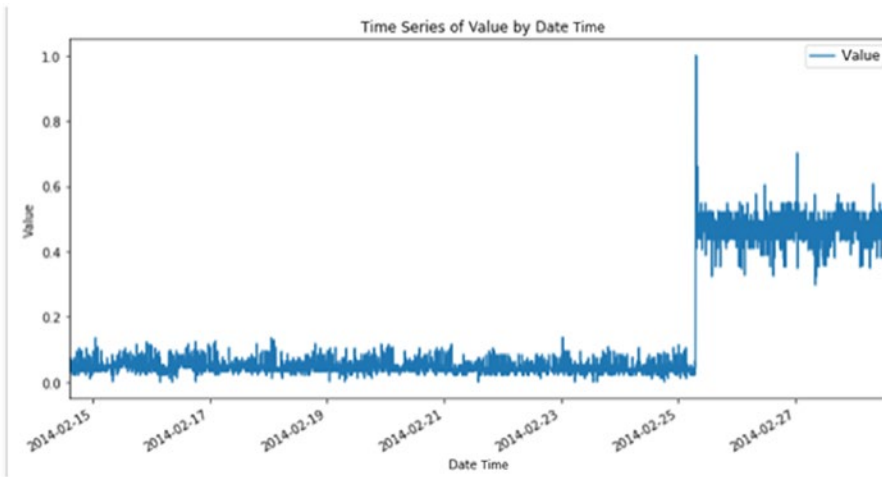


Figure 6-53. A graph showing the time series

Let's add the anomaly column to the original dataframe and prepare a new dataframe. Using visualization, you can plot the new time series now. As shown below, the time series shows the datetime vs. the value column. Normal data points are shown in green and anomalies are shown in red. Figure 6-54 shows the code to generate a graph showing anomalies.

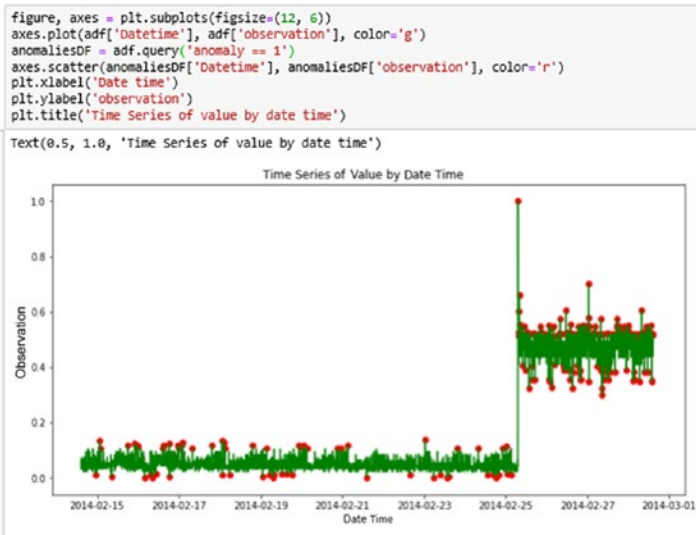


Figure 6-54. A graph showing anomalies

Since this data set has some noise or anomalies, there are anomalies (datapoints in RED) shown and everything else that is normal is green.

Summary

In this chapter, we discussed recurrent neural networks and long short-term memory models. We also looked at LSTMs as a means to detect anomalies. We also walked through several different examples of time series data with different anomalies and showed how to start detecting anomalies.

In the next chapter, we will look at another method of anomaly detection, the **temporal convolutional network**.

CHAPTER 7

Temporal Convolutional Networks

In this chapter, you will learn about temporal convolutional networks (TCNs). You will also learn how TCNs work and how they can be used to detect anomalies and how you can implement anomaly detection using a TCN.

In a nutshell, the following topics will be covered throughout this chapter:

- What is a temporal convolutional network?
- Dilated temporal convolutional networks
- Encoder-decoder temporal convolutional networks
- TCN applications

What Is a Temporal Convolutional Network?

Temporal convolutional networks refer to a family of architectures that incorporate one-dimensional convolutional layers. More specifically, these convolutions are **causal**, meaning no information from the future is leaked into the past. In other words, the model only processes information going forward in time. One of the problems with recurrent neural networks in the context of language translation is that it reads sentences from left to right in time, leading it to mistranslate in some cases where the order of the sentence is switched around to create emphasis. To solve this, bi-directional encoders were used, but this meant future information would be considered in the present. Temporal convolutional networks don't have this problem because they don't rely on information from previous time steps, unlike recurrent neural networks, thanks to their causality. Additionally, TCNs can map an input sequence of any length to an output sequence with the same length, just as a recurrent neural network (RNN) can do.

Basically, temporal convolutional networks seem to be a great alternative to RNNs. These are the advantages of TCNs, specifically considering RNNs in general:

- **Parallel computations:** Convolutional networks pair well with GPU training, particularly because the matrix-heavy calculations of the convolutional layers are well suited to the structure of GPUs, which are configured to carry out matrix calculations that are part of graphics processing. Because of this, TCNs can train much faster than RNNs.
- **Flexibility:** TCNs can change input size, filter size, increase dilation factors, stack more layers, etc. in order to easily be applied to various domains.
- **Consistent gradients:** Because TCNs are comprised of convolutional layers, they backpropagate differently than RNNs do, and thus all of the gradients are saved. RNNs have a problem called exploding or vanishing gradients, where sometimes the calculated gradient is either extremely large or extremely small, leading to the readjusted weight to be too extreme of a change or to be a relatively nonexistent change. To combat this, types of RNNs such as the LSTM, GRU, and HF-RNN, were developed.
- **Lighter on memory:** LSTMs store information in their cell gates so if the input sequence is long, much more memory is used by the LSTM network. Comparatively, TCNs are relatively straightforward because they are comprised of several layers that all share their own respective filters. Compared to LSTMs, TCNs are much lighter to run in regards to their memory usage.

However, TCNs do carry some disadvantages:

- **Memory usage during evaluation mode:** RNNs only need to know some input x_t to generate a prediction, since they maintain a summary of everything they learned through their hidden state vectors. In comparison, TCNs need the entire sequence up until the current point again to make an evaluation, leading to potentially higher memory usage than an RNN.

- **Problems with transfer learning:** First, let's define what **transfer learning** is. **Transfer learning** is when a model has been trained for one particular task (classifying vehicles for example), and has the last layer(s) taken out and retrained completely so that the model can be used for a new classification task (classifying animals, for example).

In computer vision, there are some really powerful models, such as the inception-v3 model, that have been trained on powerful GPUs for quite some time in order to achieve the performances that they do. Instead of training our own CNN from the ground up (and most of us don't have the GPU hardware or the time to spend in long training an extremely deep model like inception-v3), we can simply take inception-v3, for example, which is really good at extracting features out of images, and train it to associate the features that it extracts with a completely new set of classes. This process takes a lot less time since the weights in the entire network are already well optimized, so you're only concerned with finding the optimal weights for the layers you are retraining.

That's why transfer learning is such a valuable process; it allows us to take a pretrained, high-performance model and simply retrain the last layer(s) with our hardware and teach the model a new classification task (for CNNs).

Going back to TCNs, the model might be required to remember varying levels of sequence history in order to make predictions. If the model did not have to take in as much history in the old task to make predictions, but in the new task it had to receive even more/less history to make predictions, that would cause issues and might lead the model to perform poorly.

In a one-dimensional convolutional layer, we still have parameter **k** to determine the size of our kernel, or filter. The way the convolutional layer works is pretty similar to the two-dimensional convolutional layer you looked at in Chapter 3, but we are only dealing with vectors in this case.

Here's an example of what the one-dimensional convolutional operation looks like. Assuming an input vector defined as in Figure 7-1,

$$x = \begin{bmatrix} 10 & 5 & 15 & 20 & 10 & 20 \end{bmatrix}$$

Figure 7-1. A vector x defined with these corresponding values. This is the input vector

and a filter initialized as in Figure 7-2,

$$\begin{array}{c} \text{Filter Weights} \\ \begin{bmatrix} 1 & 0.2 & 0.1 \end{bmatrix} \end{array}$$

Figure 7-2. The filter weights associated with this one-dimensional convolutional layer

the output of the convolutional layer is calculated as shown in Figure 7-3, Figure 7-4, Figure 7-5, and Figure 7-6.

$$\begin{array}{rcl} \text{Input:} & \begin{bmatrix} 10 & 5 & 15 & 20 & 10 & 20 \end{bmatrix} \\ * & & \\ \text{Filter Weights:} & \begin{bmatrix} 1 & 0.2 & 0.1 \end{bmatrix} \\ \parallel & & \\ & 10 + 1 + 1.5 \\ & \downarrow \\ \text{Output:} & \begin{bmatrix} 12.5 \end{bmatrix} \end{array}$$

Figure 7-3. How the first entry of the output vector is calculated using the filter weights. The filter weights are multiplied element-wise with the first three entries in the input, and the results are summed up to produce the output value

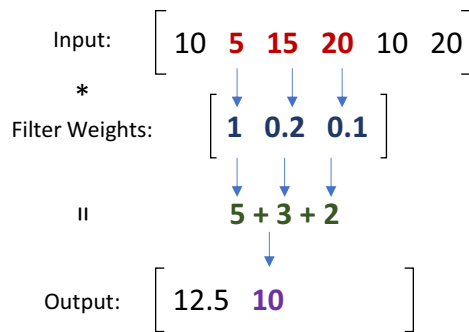


Figure 7-4. How the second entry of the output vector is calculated using the filter weights. The procedure is the same as in Figure 7-3, but the filter weights are shifted right one

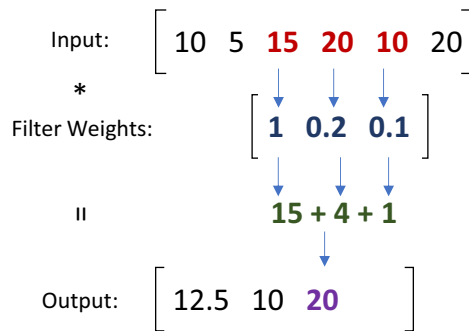


Figure 7-5. How the third entry of the output vector is calculated using the filter weights

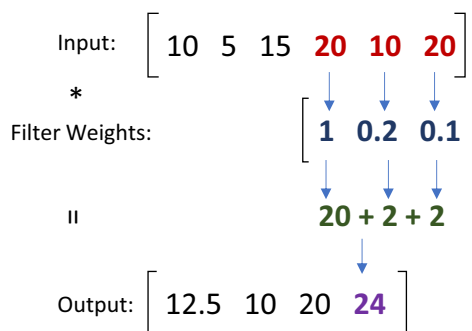


Figure 7-6. How the last entry of the output vector is calculated using the filter weights

Now we have the output of the one-dimensional convolutional layer. These one-dimensional convolutional layers are quite similar to how two-dimensional convolutional layers work, and they comprise nearly the entirety of the two different TCNs we will look at: the **dilated temporal convolutional network** and the **encoder-decoder based temporal convolutional network**. It is important to note that both models involve **supervised anomaly detection**, although the encoder-decoder TCN is capable of semi-supervised anomaly detection since it is an autoencoder.

Dilated Temporal Convolutional Network

In this type of TCN, we deal with a new property known as a **dilation**. Basically, when the dilation factor is greater than 1, we introduce gaps in the output data that correspond to the dilation factor. To understand the concept of dilation better, let's look at how it works for a **two-dimensional convolutional layer**.

This is a standard convolution, equivalent to what you looked at in Chapter 3. You can also think of a standard convolutional layer as having a dilation factor of **one** (refer to Figure 7-7).

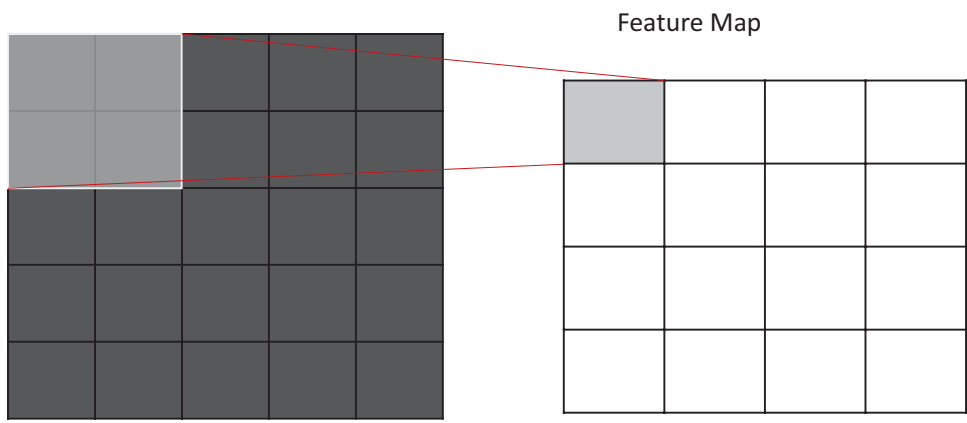


Figure 7-7. A standard convolution with a dilation factor of one

Now, let's look at what happens when we increase the **dilation factor** to two. For the first entry in the feature map, the convolution looks like Figure 7-8.

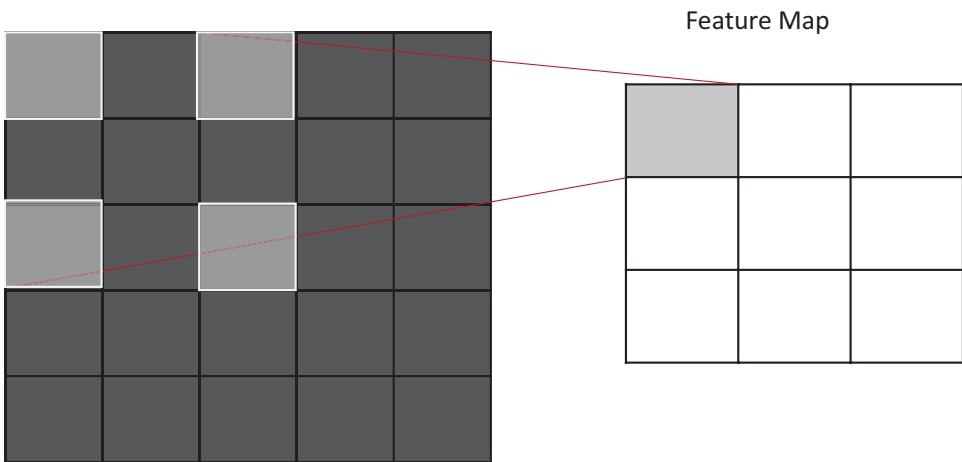


Figure 7-8. A standard convolution with a dilation factor of two defining the first entry in the feature map

Notice that the spacing between each sampled entry has increased by one across all directions. Vertically, horizontally, and diagonally, the sampled entries are all spaced apart by one entry. Essentially, this spacing is determined by finding what $\mathbf{d} - 1$ is, where $\mathbf{d}$ is the **dilation factor**. For a dilation factor of three, this spacing will be two apart. Now, for the second entry, the convolution process proceeds as normal (see Figure 7-9).

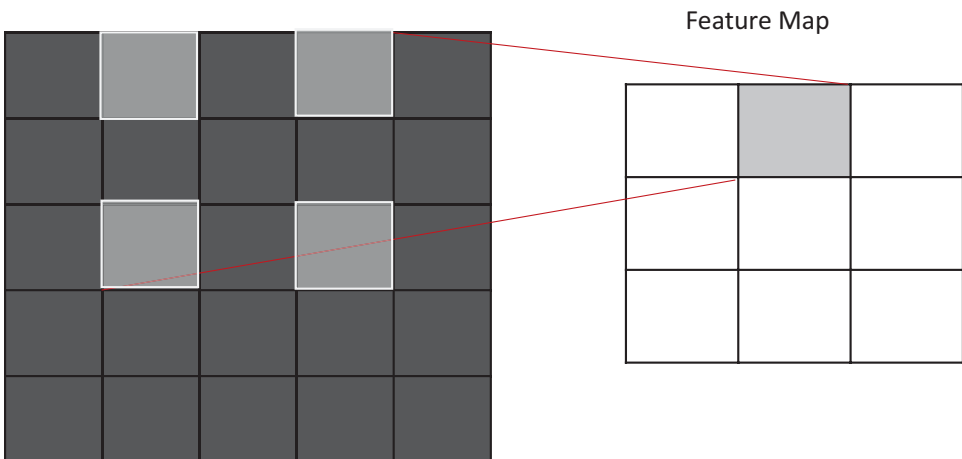


Figure 7-9. The convolution with a dilation factor of two defining the second entry in the feature map

Once the process terminates, we will have our feature map. Notice the reduction in dimensionality of the feature map, which is a direct result of increasing the dilation factor. In the standard two-dimensional convolutional layer, we had a 4x4 feature map since the dilation factor was one, but now we have a 3x3 feature map after increasing this factor to two.

A one-dimensional dilated convolution is similar. Let’s revisit the one-dimensional convolution example and modify it a bit to illustrate this concept.

Assume now that the new input vector and filter weights are as shown in Figure 7-10 and Figure 7-11.

$$x = \begin{bmatrix} 2 & 8 & 12 & 4 & 6 & 4 & 2 & 12 \end{bmatrix}$$

Figure 7-10. The new input vector weights

and

Filter Weights

$$\begin{bmatrix} 0.5 & 0.2 & 0.4 \end{bmatrix}$$

Figure 7-11. The new filter weights

Let’s also assume now that the dilation factor is two, not one. The new output vector is the following, using dilated one-dimensional convolutions with a dilation factor of two (see Figure 7-12, Figure 7-13, Figure 7-14, and Figure 7-15).

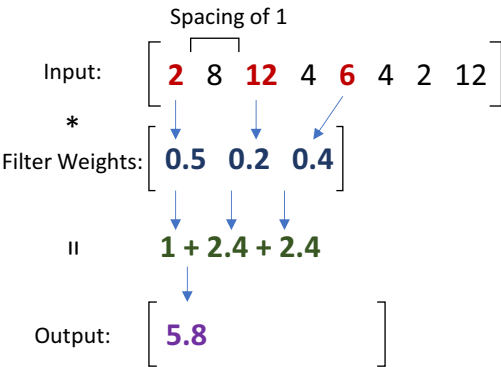


Figure 7-12. Calculating the first entry in the output factor using dilated one-dimensional convolutions with a dilation factor of two

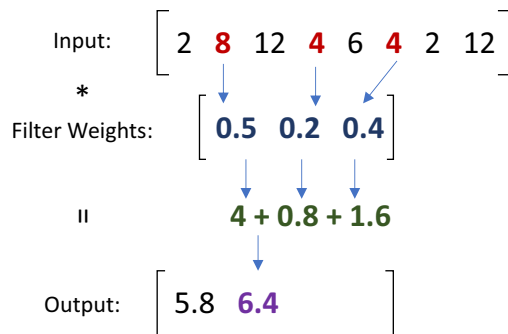


Figure 7-13. The next set of three input vector values are multiplied with the filter weights to produce the next output vector value

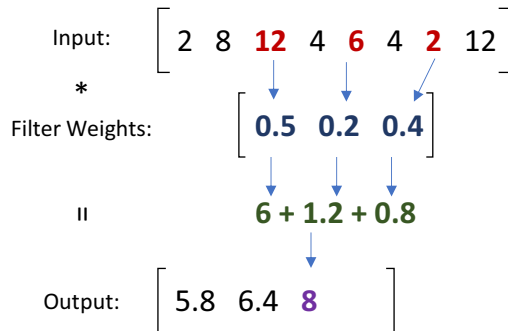


Figure 7-14. The third set of three input vector values are multiplied with the filter weights to produce the next output vector value

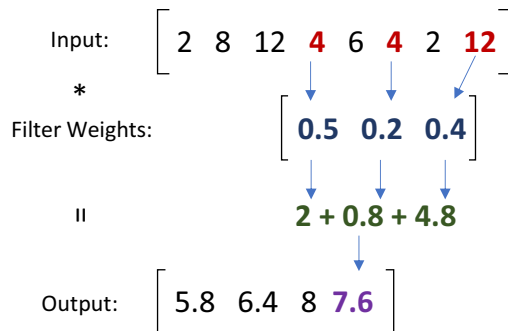


Figure 7-15. The final set of three input vector values are multiplied with the filter weights to produce the last output vector value

Now that we’ve covered what a **dilated convolution** looks like in the context of one-dimensional convolutions, let’s look at the difference between an **acausal** and a **casual** dilated convolution. To illustrate this concept, assume that both examples are referring to a set of dilated one-dimensional convolutional layers. With that in mind, Figure 7-16 shows an **acausal** network.

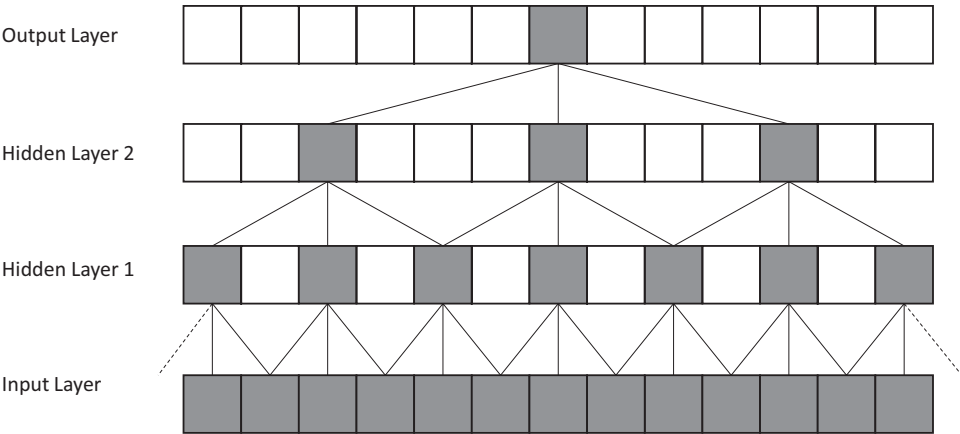


Figure 7-16. An acausal dilated network. The first hidden layer has a dilation factor of two, and the second hidden layer has a dilation factor of four. Notice how inputs “forward in the sequence” contribute to the next layer’s node as well

It might not be that apparent from the way the architecture is structured, but if you think of the input layer as a sequence of some data going forward in time, you might be able to see that information from the future would be accounted for when selecting the output. In a **casual** network, we only want information that we’ve learned up until the present, so none of the information from the future will be accounted for in the model’s predictions. Figure 7-17 shows what a **casual** network looks like.

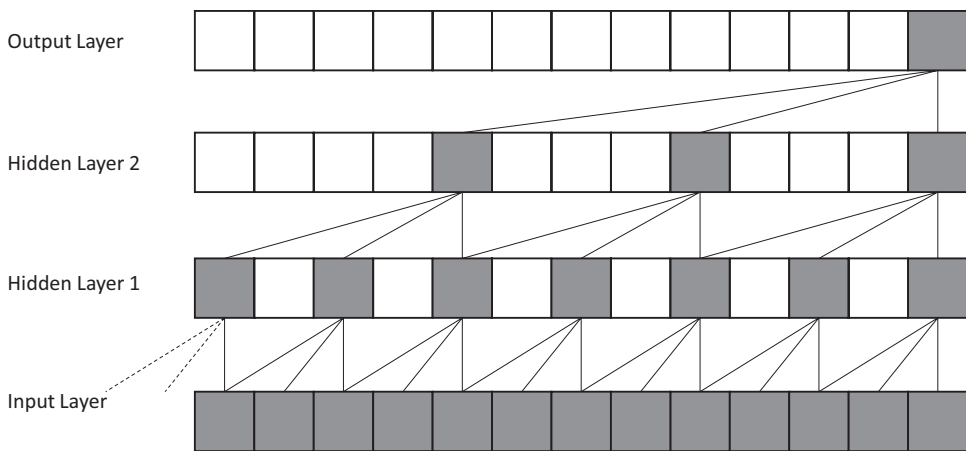


Figure 7-17. A causal dilated network. The first hidden layer has a dilation factor of two, and the second hidden layer has a dilation factor of four. Notice how no inputs forward in the sequence contribute to the next layer's node. This type of structure is ideal if the goal is to preserve some sort of flow within the data set, which is time in our case

From this, we can see how the linear nature of time is preserved in the model, and how no information from the future would be learned by the model. In casual networks, only information from the past until the present is considered by the model. The dilated temporal convolutional network we are referring to has a similar model architecture, utilizing dilated causal convolutions in each layer preceding the output layer.

Anomaly Detection with the Dilated TCN

Now that you know more about what a TCN is and how it works, let's try applying a dilated TCN to the credit card dataset.

First, import all of the necessary packages (see Figure 7-18a).

```

import keras
from keras import regularizers, optimizers
from keras import losses
from keras.models import Sequential, Model, load_model
from keras.layers import Dense, Input, Dropout, Embedding, LSTM
from keras.optimizers import RMSprop, Adam, Nadam
from keras.preprocessing import sequence

from keras.layers import Conv1D, Flatten, Activation, SpatialDropout1D
from keras.callbacks import ModelCheckpoint, TensorBoard
from keras.utils import to_categorical

import sklearn
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, roc_auc_score
from sklearn.metrics import classification_report

import seaborn as sns
import pandas as pd
import numpy as np
import matplotlib

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
%matplotlib inline

import tensorflow
import sys
print("Python: ", sys.version)

print("pandas: ", pd.__version__)
print("numpy: ", np.__version__)
print("seaborn: ", sns.__version__)
print("matplotlib: ", matplotlib.__version__)
print("sklearn: ", sklearn.__version__)
print("Keras: ", keras.__version__)
print("Tensorflow: ", tensorflow.__version__)

```

```

Python: 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)]
pandas: 0.24.2
numpy: 1.16.4
seaborn: 0.9.0
matplotlib: 3.1.0
sklearn: 0.21.2
Keras: 2.2.4
Tensorflow: 1.13.1

```

Figure 7-18a. Importing all of the necessary packages in order to start your code

Then, you must create a class for the visualization of confusion matrix, etc. (see Figure 7-18b).

```

class Visualization:
    labels = ["Normal", "Anomaly"]

    def draw_confusion_matrix(self, y, ypred):
        matrix = confusion_matrix(y, ypred)

        plt.figure(figsize=(10, 8))
        colors = ["orange", "green"]
        sns.heatmap(matrix, xticklabels=self.labels, yticklabels=self.labels, cmap=colors, annot=True,
                    plt.title("Confusion Matrix")
                    plt.ylabel('Actual')
                    plt.xlabel('Predicted')
                    plt.show()

    def draw_anomaly(self, y, error, threshold):
        groupsDF = pd.DataFrame({'error': error,
                                'true': y}).groupby('true')

        figure, axes = plt.subplots(figsize=(12, 8))

        for name, group in groupsDF:
            axes.plot(group.index, group.error, marker='x' if name == 1 else 'o', linestyle='',
                    color='r' if name == 1 else 'g', label="Anomaly" if name == 1 else "Normal")

        axes.hlines(threshold, axes.get_xlim()[0], axes.get_xlim()[1], colors="b", zorder=100, label='')
        axes.legend()

        plt.title("Anomalies")
        plt.ylabel("Error")
        plt.xlabel("Data")
        plt.show()

    def draw_error(self, error, threshold):
        plt.plot(error, marker='o', ms=3.5, linestyle='',
                label='Point')

        plt.hlines(threshold, xmin=0, xmax=len(error)-1, colors="b", zorder=100, label='Threshold')
        plt.legend()
        plt.title("Reconstruction error")
        plt.ylabel("Error")
        plt.xlabel("Data")
        plt.show()

```

Figure 7-18b. *Creating a visualization class*

After that, proceed to importing your data set and processing it (see Figure 7-19).

```

df = pd.read_csv("datasets/creditcardfraud/creditcard.csv",
                sep=",", index_col=None)

print(df.shape)

df.head()

```

Figure 7-19. *Importing your data set and displaying the first five entries*

The output should look somewhat like Figure 7-20.

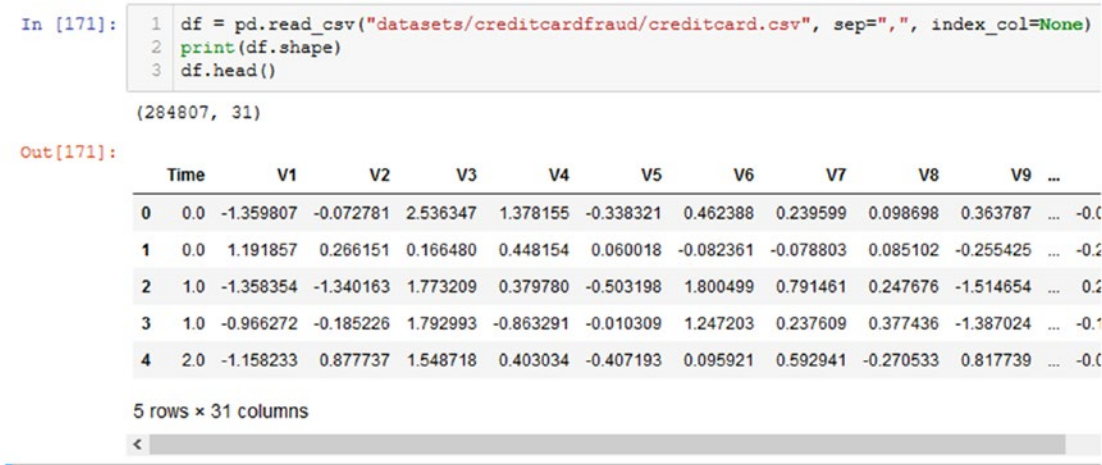


Figure 7-20. The first five entries of the data frame

The data frame continues in Figure 7-21.



Figure 7-21. The output in Figure 7-20 scrolled right

Each entry is noticeably large, with 31 columns per entry. If you check the tail end of the data frame in Figure 7-22,

```
In [154]: 1 df.tail()
```

```
Out[154]:
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.305334	1.914428	...	C
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294869	0.584800	...	C
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.708417	0.432454	...	C
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.679145	0.392087	...	C
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.414650	0.486180	...	C

5 rows × 31 columns

Figure 7-22. The tail end of the data frame. Notice how large the values for time get

you can see that the data set is pretty massive with 284,807 entries in total (the index starts at 0). Additionally, notice how the values for time become absurdly large. If you pass in values this large into the model for training, you are bound to get errors with convergence. Not only that, it's just good practice to normalize any large values, since it improves performance and training efficiency if you pass in smaller values to the model. Run the code in Figure 7-23 to standardize the values for Time and for Amount.

```
df['Amount'] =
StandardScaler().fit_transform(df['Amount'].values.reshape(-1, 1))

df['Time'] = StandardScaler().fit_transform(df['Time'].values.reshape(-
1, 1))

df.tail()
```

Figure 7-23. This code standardizes the values for Time and Amount

Now you can see that the values for the columns Time (Figure 7-24)

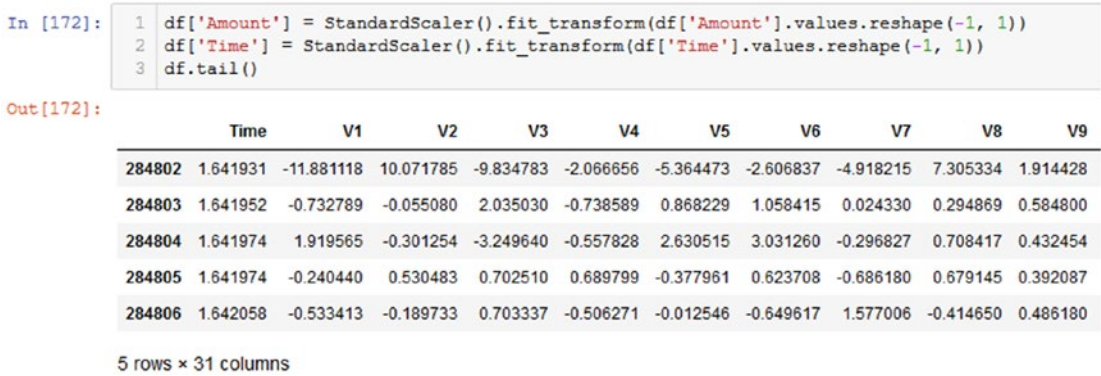


Figure 7-24. The standardized values for the Time column
and for Amount (Figure 7-25)

...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
...	0.213454	0.111864	1.014480	-0.509348	1.436807	0.250034	0.943651	0.823731	-0.350151	0
...	0.214205	0.924384	0.012463	-1.016226	-0.606624	-0.395255	0.068472	-0.053527	-0.254117	0
...	0.232045	0.578229	-0.037501	0.640134	0.265745	-0.087371	0.004455	-0.026561	-0.081839	0
...	0.265245	0.800049	-0.163298	0.123205	-0.569159	0.546668	0.108821	0.104533	-0.313249	0
...	0.261057	0.643078	0.376777	0.008797	-0.473649	-0.818267	-0.002415	0.013649	0.514355	0

Figure 7-25. The standardized values for the Amount column

are much smaller and much more manageable numbers to pass in.

Since there are so many entries in the entire data set, it's best to limit the number of "normal" data entries you feed into the model since the model seems to ignore the anomalies if the entire data set is passed in. To avoid drowning out the anomalous data entries, let's pick 10,000 normal entries to derive your training and testing data sets from (see Figure 7-26).

```
anomalies = df[df["Class"] == 1]
normal = df[df["Class"] == 0]

anomalies.shape, normal.shape
```

Figure 7-26. Defining two data frames: anomalies and normal

The output should look somewhat like Figure 7-27.


```
In [5]: 1 anomalies = df[df["Class"] == 1]
        2 normal = df[df["Class"] == 0]
        3
        4 anomalies.shape, normal.shape
        5

Out[5]: ((492, 31), (284315, 31))
```

Figure 7-27. The output of the code in Figure 7-26

In this block of code, you name two new data frames as `anomalies` and `normal`, with their names corresponding to their content. Checking their shape reveals that there are relatively few anomalies compared to the entire data set, comprising around 0.173% of the whole data set.

Now let's get to defining your training and testing data sets (see Figure 7-28).

```
for f in range(0, 20):
    normal = normal.iloc[np.random.permutation(len(normal))]

data_set = pd.concat([normal[:2000], anomalies])

x_train, x_test = train_test_split(data_set, test_size = 0.4,
    random_state = 42)

x_train = x_train.sort_values(by=['Time'])
x_test = x_test.sort_values(by=['Time'])

y_train = x_train["Class"]
y_test = x_test["Class"]

x_train.head(10)
```

Figure 7-28. Defining the training and testing sets and sorting both by time to maintain the temporal flow

Shuffling the normal data set as well as using the `train_test_split` function to randomly select testing and training samples helps ensure that you pick a good range of data values to represent normal data. You can limit the number of iterations in the `for` block at the start of the code if you wish.

From there, the first 10,000 data entries of the shuffled normal data are concatenated with the anomalies, and the training and testing data sets are created. Both sets are then sorted by the `Time` column to maintain the entire aspect of time.

The output should look somewhat like the Figure 7-29.

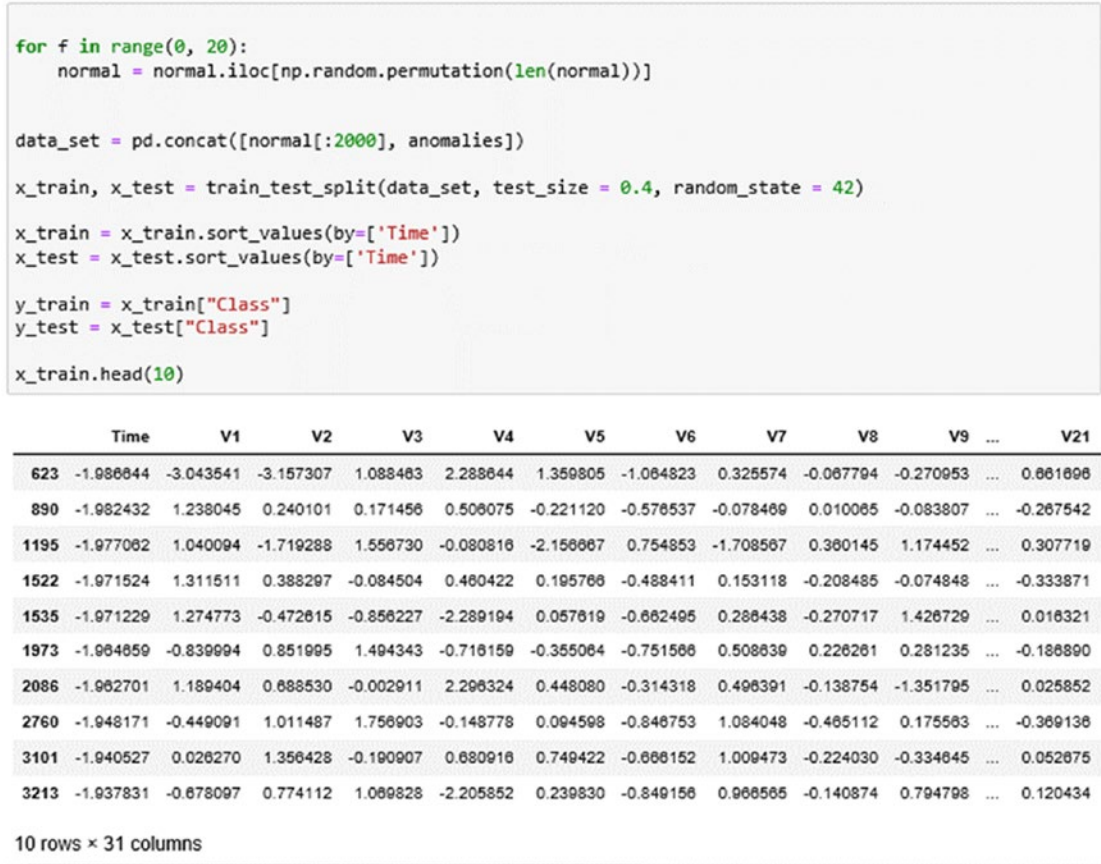


Figure 7-29. The data sets sorted by the `Time` column

Notice how the indices vary in number, although they are all ordered by time. Now you can move on to reshaping your data sets to pass into the model. Running the code block in Figure 7-30 can give you a sense of how the data sets are structured.

```
print("Shapes:\nx_train:%s\ny_train:%s\n" % (x_train.shape,
y_train.shape))

print("x_test:%s\ny_test:%s\n" % (x_test.shape,
y_test.shape))
```

Figure 7-30. *Outputs the shapes to provide an understanding of how the data sets are structured*

The output should look somewhat like Figure 7-31.

```
print("Shapes:\nx_train:%s\ny_train:%s\n" % (x_train.shape, y_train.shape))
print("x_test:%s\ny_test:%s\n" % (x_test.shape, y_test.shape))

Shapes:
x_train:(1495, 31)
y_train:(1495,)

x_test:(997, 31)
y_test:(997,)
```

Figure 7-31. *The shapes of both data sets*

To pass the data sets into the model, the x sets must be three-dimensional, and the y sets must be two-dimensional. You can simply reshape the x sets, and change the y sets to be categorical (refer to Chapter 3 to see what the `keras.to_categorical()` function does).

Run the code in Figure 7-32.

```
x_train = np.array(x_train).reshape(x_train.shape[0],
x_train.shape[1], 1)

x_test = np.array(x_test).reshape(x_test.shape[0],
x_test.shape[1], 1)

input_shape = (x_train.shape[1], 1)

y_train = keras.utils.to_categorical(y_train, 2)
y_test = keras.utils.to_categorical(y_test, 2)
```

Figure 7-32. *Makes the x sets three-dimensional and the y sets two-dimensional by reshaping the x sets and changing the y sets to be categorical. The reshaping of the x sets is done to fit the input shape of the model*

Let's take a look at how the operations changed the data sets. Run the code in Figure 7-33.

```
print("Shapes:\nx_train:%s\ny_train:%s\n" % (x_train.shape,
y_train.shape))

print("x_test:%s\ny_test:%s\n" % (x_test.shape, y_test.shape))

print("input_shape:{}\n".format(input_shape))
```

Figure 7-33. Code to print the shapes of the data sets to see how the operations changed the structure

The output should look like Figure 7-34.

```
print("Shapes:\nx_train:%s\ny_train:%s\n" % (x_train.shape, y_train.shape))
print("x_test:%s\ny_test:%s\n" % (x_test.shape, y_test.shape))
print("input_shape:{}\n".format(input_shape))

Shapes:
x_train:(1495, 31, 1)
y_train:(1495, 2)

x_test:(997, 31, 1)
y_test:(997, 2)

input_shape:(31, 1)
```

Figure 7-34. The *x* sets are three-dimensional while the *y* sets are two-dimensional

Alright, now both of the data sets have been reshaped successfully. The input shape tells the model how many columns and rows to accept per entry. In this case, the input shape indicates that there will be 1 row and 31 columns.

Now let's move on to defining your model. The code chunk in Figure 7-35 defines the one-dimensional convolutional layers and the dropout layers.

```

input_layer = Input(shape=(input_shape ))

#Series of temporal convolutional layers with dilations increasing by
powers of 2.
conv_1 = Conv1D(filters=128, kernel_size=2, dilation_rate=1,
                padding='causal', strides=1, input_shape=input_shape,
                kernel_regularizer=regularizers.l2(0.01),
                activation='relu')(input_layer)

#Dropout layer after each 1D-convolutional layer
drop_1 = SpatialDropout1D(0.05)(conv_1)

conv_2 = Conv1D(filters=128, kernel_size=2, dilation_rate=2,
                padding='causal', strides=1,
                kernel_regularizer=regularizers.l2(0.01),
                activation='relu')(drop_1)

drop_2 = SpatialDropout1D(0.05)(conv_2)

conv_3 = Conv1D(filters=128, kernel_size=2, dilation_rate=4,
                padding='causal',
                strides=1, kernel_regularizer=regularizers.l2(0.01),
                activation='relu')(drop_2)

drop_3 = SpatialDropout1D(0.05)(conv_3)

conv_4 = Conv1D(filters=128, kernel_size=2, dilation_rate=8,
                padding='causal',
                strides=1, kernel_regularizer=regularizers.l2(0.05),
                activation='relu')(drop_3)

drop_4 = SpatialDropout1D(0.05)(conv_4)

```

Figure 7-35. Defines all of the one-dimensional convolutional layers and the dropout layers in the model

The code chunk in Figure 7-36 defines the last two layers, which consist of a layer to flatten the data and one layer to represent the two classes.

```
#Flatten layer to feed into the output layer
flat = Flatten()(drop_4)

output_layer = Dense(2, activation='softmax')(flat)

TCN = Model(inputs=input_layer, outputs=output_layer)
```

Figure 7-36. Defines the last two layers, which consist of a layer to flatten the data and one layer to represent the two classes

Now let's compile the model and look at the summary of the layers (see Figure 7-37).

```
TCN.compile(loss=keras.losses.categorical_crossentropy,
             optimizer=optimizers.Adam(lr=0.002),
             metrics=['mae', 'accuracy'])

checkpointer = ModelCheckpoint(filepath="model_TCN_creditcard.h5",
                               verbose=0,
                               save_best_only=True)

TCN.summary()
```

Figure 7-37. Code to compile the data, define a callback to save the model under the given filepath, and output the summary of the model

The output should look like Figure 7-38.

```
TCN.compile(loss=keras.losses.categorical_crossentropy,
             optimizer=optimizers.Adam(lr=0.002),
             metrics=['mae', 'accuracy'])

checkpointer = ModelCheckpoint(filepath="model_TCN_creditcard.h5",
                               verbose=0,
                               save_best_only=True)

TCN.summary()
```

Layer (type)	Output Shape	Param #
input_11 (InputLayer)	(None, 31, 1)	0
conv1d_41 (Conv1D)	(None, 31, 128)	384
spatial_dropout1d_41 (SpatialDropout1D)	(None, 31, 128)	0
conv1d_42 (Conv1D)	(None, 31, 128)	32896
spatial_dropout1d_42 (SpatialDropout1D)	(None, 31, 128)	0
conv1d_43 (Conv1D)	(None, 31, 128)	32896
spatial_dropout1d_43 (SpatialDropout1D)	(None, 31, 128)	0
conv1d_44 (Conv1D)	(None, 31, 128)	32896
spatial_dropout1d_44 (SpatialDropout1D)	(None, 31, 128)	0
flatten_11 (Flatten)	(None, 3968)	0
dense_11 (Dense)	(None, 2)	7938
Total params: 107,010		
Trainable params: 107,010		
Non-trainable params: 0		

Figure 7-38. The summary of the model. You can use this to help debug your models when you're creating one from scratch by checking that the output shapes for the layers match the input shapes of the subsequent layer

Looking at the model summary can help you understand more about what's going on at each layer. Sometimes, it can help with debugging, where there can be dimensionality reductions that you don't expect. For example, sometimes when odd dimensions become reduced by a factor of 2, they might become rounded down. When expanding back up, this can prove to be problematic because the new dimension does not match the old dimension. You can expect to run into problems like these with autoencoders, where the entire aim of the architecture is to compress the data and attempt to reconstruct it.

Run the code in Figure 7-39 to begin the training process.

```
TCN.fit(x_train, y_train,
        batch_size=128,
        epochs=25,
        verbose=1,
        validation_data=(x_test, y_test),
        callbacks = [checkpointer])
```

Figure 7-39. Code to start the training process for the model

You should see something like Figure 7-40 during the training process.

```
In [169]: 1 TCN.fit(x_train, y_train,
2             batch_size=128,
3             epochs=25,
4             verbose=1,
5             validation_data=(x_test, y_test),
6             callbacks = [checkpointer])
7
```

```
Train on 6295 samples, validate on 4197 samples
Epoch 1/25
6295/6295 [=====] - 4s - loss: 3.1321 - acc: 0.9633 - val_loss: 0.3182 - val_acc: 0.9881
Epoch 2/25
6295/6295 [=====] - 0s - loss: 0.1426 - acc: 0.9873 - val_loss: 0.0958 - val_acc: 0.9888
Epoch 3/25
6295/6295 [=====] - 0s - loss: 0.0857 - acc: 0.9889 - val_loss: 0.0809 - val_acc: 0.9909
Epoch 4/25
6295/6295 [=====] - 0s - loss: 0.0716 - acc: 0.9900 - val_loss: 0.0748 - val_acc: 0.9909
Epoch 5/25
6295/6295 [=====] - 0s - loss: 0.0722 - acc: 0.9897 - val_loss: 0.0728 - val_acc: 0.9909
Epoch 6/25
6295/6295 [=====] - 0s - loss: 0.0669 - acc: 0.9906 - val_loss: 0.0826 - val_acc: 0.9852
```

Figure 7-40. The output during the training process

At the end, you should see something like Figure 7-41.

```
Epoch 21/25
6295/6295 [=====] - 0s - loss: 0.0566 - acc: 0.9916 - val_loss: 0.0651 - val_acc: 0.9890
Epoch 22/25
6295/6295 [=====] - 0s - loss: 0.0575 - acc: 0.9914 - val_loss: 0.0652 - val_acc: 0.9878
Epoch 23/25
6295/6295 [=====] - 0s - loss: 0.0571 - acc: 0.9909 - val_loss: 0.0701 - val_acc: 0.9912
Epoch 24/25
6295/6295 [=====] - 0s - loss: 0.0570 - acc: 0.9914 - val_loss: 0.0650 - val_acc: 0.9907
Epoch 25/25
6295/6295 [=====] - 0s - loss: 0.0584 - acc: 0.9913 - val_loss: 0.0700 - val_acc: 0.9874
```

Figure 7-41. The output when the training process ends

Now that the training is finished, you can evaluate your model's performance (see Figure 7-42).


```

score = TCN.evaluate(x_test, y_test,
                     verbose=0)

print('Test loss:', score[0])
print('Test mae:', score[1])
print('Test accuracy:', score[2])

```

Figure 7-42. Code to evaluate the loss and the accuracy on the test sets

The output should look somewhat like Figure 7-43.

```

score = TCN.evaluate(x_test, y_test, verbose=1)
print('Test loss:', score[0])
print('Test mae:', score[1])
print('Test accuracy:', score[2])

997/997 [=====] - 0s 101us/step
Test loss: 0.17798992889814655
Test mae: 0.06778481965109243
Test accuracy: 0.9648946840521565

```

Figure 7-43. The generated loss and accuracy scores for the test set. The accuracy is really good, but again, accuracy isn't always the best metric to judge models by

Now you can check the AUC score (see Figure 7-44).

```

from sklearn.metrics import roc_auc_score

preds = TCN.predict(x_test)

y_pred = np.round(preds)

auc = roc_auc_score(y_pred, y_test)

print("AUC: {:.2%}".format(auc))

```

Figure 7-44. Code to generate an AUC score given the test sets and the predictions

The output should look somewhat like Figure 7-45a.

```
from sklearn.metrics import roc_auc_score

preds = TCN.predict(x_test)
y_pred = np.round(preds)
auc = roc_auc_score( y_pred, y_test)
print("AUC: {:.2%}".format (auc))
```

AUC: 97.41%

Figure 7-45a. The generated AUC score of 99.02% for this model

For the classification report and confusion matrix, see Figure 7-45b.

```
print(classification_report(y_test, y_pred))
```

	precision	recall	f1-score	support
0	0.96	1.00	0.98	799
1	0.99	0.83	0.90	198
micro avg	0.96	0.96	0.96	997
macro avg	0.97	0.92	0.94	997
weighted avg	0.97	0.96	0.96	997
samples avg	0.96	0.96	0.96	997

```
viz = Visualization()
y_pred2 = np.argmax(y_pred, axis=1)
y_test2 = np.argmax(y_test, axis=1)
viz.draw_confusion_matrix(y_test2, y_pred2)
```

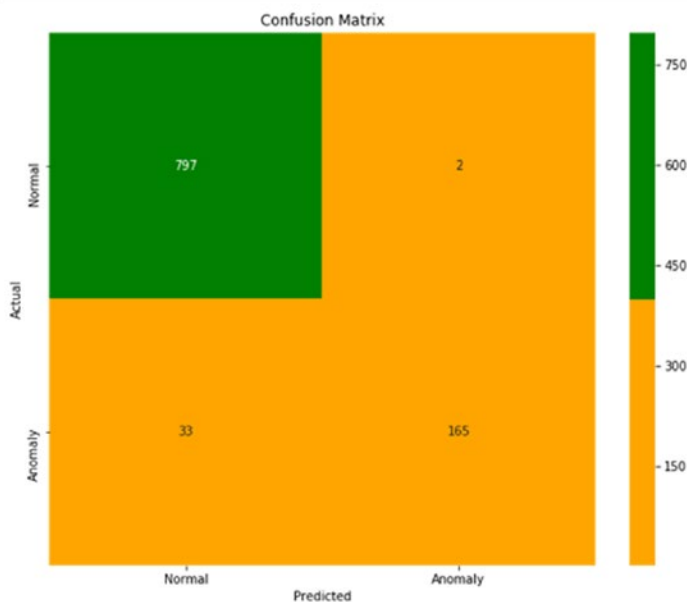


Figure 7-45b. Classification report and confusion matrix

That's a pretty good AUC score! However, this was an example of **supervised anomaly detection**, meaning you had the anomalies and the normal data labeled. You won't always have this luxury, and you shouldn't expect it either because of the massive volumes of data that can be involved. For your next example, you will be implementing the encoder-decoder based temporal convolutional network (ED-TCN), but it will also be an instance of supervised anomaly detection so that it can be compared to the dilated TCN model given a similar task. However, keep in mind that since it is based on an autoencoder framework, the ED-TCN should also be able to perform **semi-supervised anomaly detection**.

Encoder-Decoder Temporal Convolutional Network

The version of the encoder-decoder TCN you will be exploring involves a combination of one-dimensional causal convolutional and pooling layers to encompass the encoding stage and a series of upsampling and one-dimensional causal convolutional layers to comprise the decoding stage. The convolutional layers in this model aren't dilated, but they still count as layers of a temporal convolutional network. To better understand the structure of this model, take a look at [Figure 7-46](#).

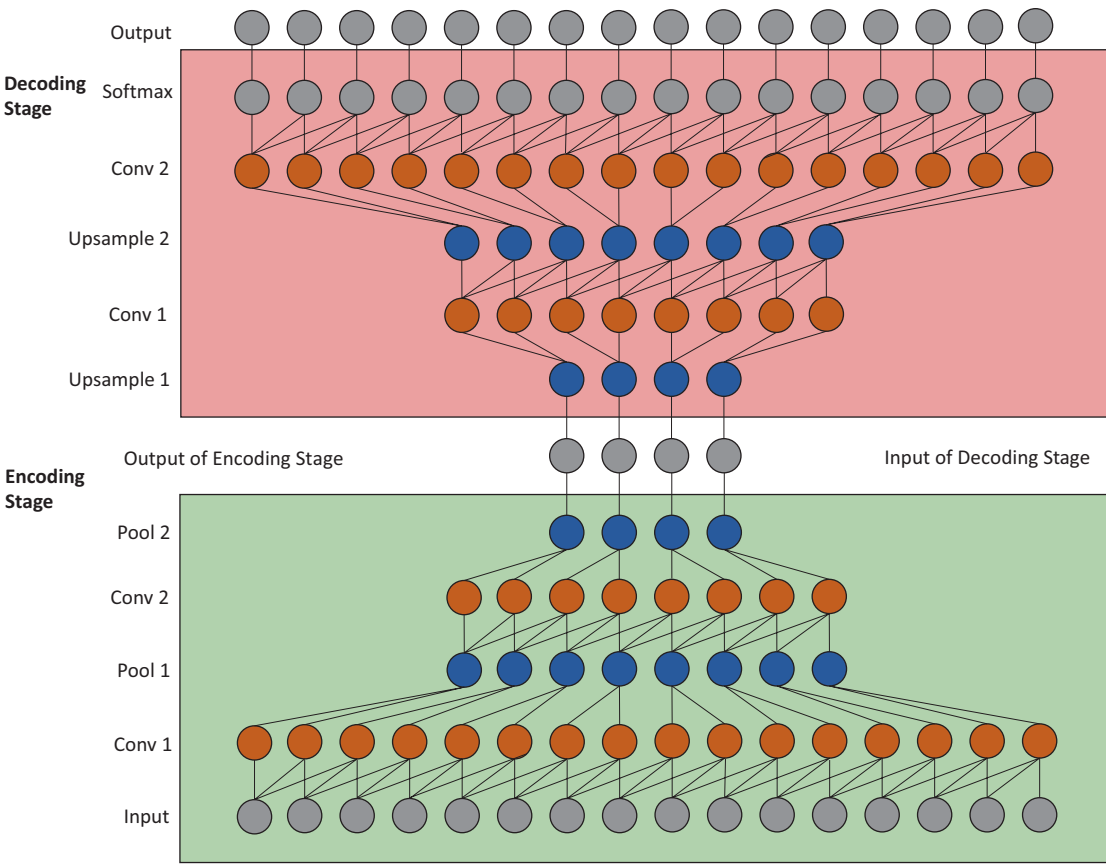


Figure 7-46. In both the encoding and decoding stages, the model is comprised of causal convolutional layers and is structured so that the layers are always causal

The diagram might seem pretty complicated, so let’s break it down layer by layer.

First, look at the encoding stage and start with the input layer at the very bottom. From this layer, you perform a **causal convolution** on the input as part of the first convolutional layer. The outputs of the first convolutional layer, which you will call **conv_1**, are now the inputs of the first **max pooling layer**, which you will call **pool_1**.

Recall from Chapter 3 that the pooling layer emphasizes the maximum value in the areas it passes through, effectively generalizing the inputs by choosing the heaviest values. From here, you have another set of causal convolutions and max pooling with layers **conv_2** and **pool_2**. Note the progressive reduction in size of the data as it passes through the encoding stage, a feature characteristic to autoencoders. Finally, you have a dense layer in the middle of the two stages, representing the final, encoded output of the encoding stage as well as the encoded input of the decoding stage.

The decoding stage is a bit different in this case, since you make use of what is called **upsampling**. Upsampling is a technique in which you repeat the data n number of times to scale it up by a factor n . In the max pooling layers, the data is reduced by a factor of two. So, to upsample and increase the data by the same factor of two, you repeat the data twice. In this case, you are using one-dimensional upsampling, so the layer repeats each step n times with respect to the axis of time. To get a better understanding of what upsampling does, let's apply one-dimensional upsampling to Figure 7-47 and Figure 7-48.

$$x = \begin{bmatrix} 4 & 2 & 6 & 7 & 1 & 6 & 9 \end{bmatrix}$$

Figure 7-47. A vector x defined with the corresponding values

$n = 2$
So data increases by factor of 2 /
repeat each step two times

Figure 7-48. The upsampling factor n

Keeping in mind that each individual temporal step is repeated twice, you would see something like Figure 7-49, Figure 7-50, and Figure 7-51.

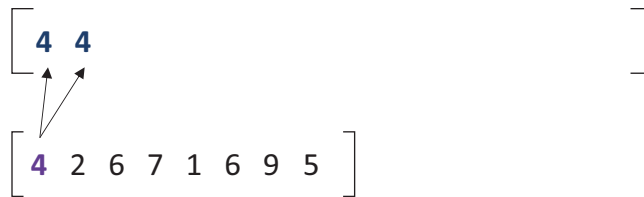


Figure 7-49. The first entry in the input is repeated twice to form the first two entries in the upsampled output vector



Figure 7-50. The next entry is repeated twice to form the next two entries in the output vector of the upsampling operation

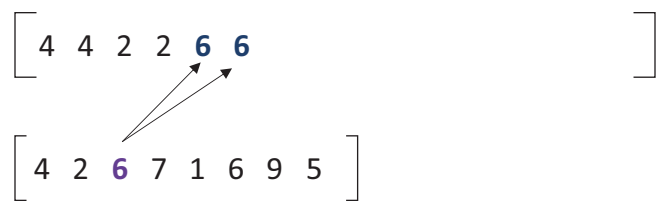


Figure 7-51. This process is repeated with the third entry in the input vector to form the next third pair of entries in the output vector

And so on until you finally get Figure 7-52.

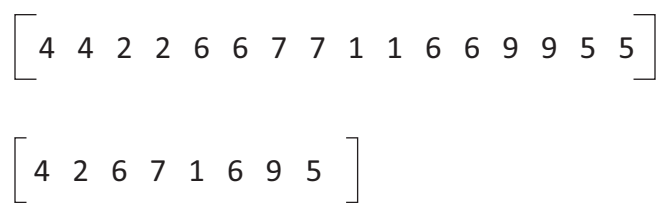


Figure 7-52. The output vector after the upsampling operation compared to the original input vector below it

Going back to the model, each upsampling layer is then connected to a one-dimensional convolutional layer, and the pair of upsampling layer and one-dimensional convolutional layer repeats again until the final output is passed through a softmax function to result in the output/prediction.

Anomaly Detection with the ED-TCN

Let’s put this model to the test by applying it to the credit card dataset. Once again, this example is another instance of **supervised learning**, so you will have both anomalies and normal data labeled.

First, begin by importing all of the necessary modules (see Figure 7-53).

```
import numpy as np
import pandas as pd
import keras

from keras import regularizers, optimizers

from keras.layers import Input, Conv1D, Dense, Flatten, Activation,
UpSampling1D, MaxPooling1D, ZeroPadding1D

from keras.callbacks import ModelCheckpoint, TensorBoard

from keras.models import Model, load_model

from keras.utils import to_categorical

from sklearn.model_selection import train_test_split

from sklearn.preprocessing.data import StandardScaler
```

Figure 7-53. *Importing the necessary modules*

Next, load your data and preprocess it. Notice that the steps are basically the same as in the first example (see [Figure 7-54](#)).

```

df['Amount'] =
StandardScaler().fit_transform(df['Amount'].values.reshape(-1, 1))
df['Time'] =
StandardScaler().fit_transform(df['Time'].values.reshape(-1, 1))

anomalies = df[df["Class"] == 1]
normal = df[df["Class"] == 0]

for f in range(0, 20):
    normal = normal.iloc[np.random.permutation(len(normal))]

data_set = pd.concat([normal[:10000], anomalies])

x_train, x_test = train_test_split(data_set, test_size = 0.4,
random_state = 42)

x_train = x_train.sort_values(by=['Time'])
x_test = x_test.sort_values(by=['Time'])

y_train = x_train["Class"]
y_test = x_test["Class"]

```

Figure 7-54. Using the standard scaler on the columns Time and Amount, defining the anomaly and normal value data sets, and then defining a new data set to generate the training and testing sets from. Finally, these sets are sorted in increasing order of time

And now you reshape the data sets as shown in Figure 7-55.

```

x_train = np.array(x_train).reshape(x_train.shape[0],
x_train.shape[1], 1)

x_test = np.array(x_test).reshape(x_test.shape[0],
x_test.shape[1], 1)

input_shape = (x_train.shape[1], 1)

y_train = keras.utils.to_categorical(y_train, 2)
y_test = keras.utils.to_categorical(y_test, 2)

```

Figure 7-55. Reshaping the training and testing sets so that they correspond with the input shape of the model

Now that the data preprocessing is done, let's build the model. This is the encoding stage (see Figure 7-56).

```
input_layer = Input(shape=(input_shape ))

### ENCODING STAGE
# Pairs of causal 1D convolutional layers and pooling layers
# comprising the encoding stage
conv_1 = Conv1D(filters=int(input_shape[0]), kernel_size=2,
                dilation_rate=1,
                padding='causal', strides=1, input_shape=input_shape,
                kernel_regularizer=regularizers.l2(0.01),
                activation='relu')(input_layer)

pool_1 = MaxPooling1D(pool_size=2, strides=2)(conv_1)

conv_2 = Conv1D(filters=int(input_shape[0] / 2), kernel_size=2,
                dilation_rate=1,
                padding='causal', strides=1,
                kernel_regularizer=regularizers.l2(0.01),
                activation='relu')(pool_1)

pool_2 = MaxPooling1D(pool_size=2, strides=3)(conv_2)

conv_3 = Conv1D(filters=int(input_shape[0] / 3), kernel_size=2,
                dilation_rate=1,
                padding='causal',
                strides=1, kernel_regularizer=regularizers.l2(0.01),
                activation='relu')(pool_2)

### OUTPUT OF ENCODING STAGE
encoder = Dense(int(input_shape[0] / 6), activation='relu')(conv_3)
```

Figure 7-56. *Defining the code for the encoding stage*

Following that block is the code for the decoding stage (see Figure 7-57).

```

### DECODING STAGE
# Pairs of upsampling and causal 1D convolutional layers comprising
the decoding stage
upsample_1 = UpSampling1D(size=3)(encoder)

conv_4 = Conv1D(filters=int(input_shape[0]/3), kernel_size=2,
dilation_rate=1,
                padding='causal', strides=1,
kernel_regularizer=regularizers.l2(0.01),
                activation='relu')(upsample_1)

upsample_2 = UpSampling1D(size=2)(conv_4)

conv_5 = Conv1D(filters=int(input_shape[0]/2), kernel_size=2,
dilation_rate=1,
                padding='causal',
strides=1, kernel_regularizer=regularizers.l2(0.05),
                activation='relu')(upsample_2)

zero_pad_1 = ZeroPadding1D(padding=(0,1))(conv_5)

conv_6 = Conv1D(filters=int(input_shape[0]), kernel_size=2,
dilation_rate=1,
                padding='causal',
strides=1, kernel_regularizer=regularizers.l2(0.05),
                activation='relu')(zero_pad_1)

### Output of decoding stage flattened and passed through softmax to
make predictions
flat = Flatten()(conv_6)

output_layer = Dense(2, activation='softmax')(flat)

TCN = Model(inputs=input_layer, outputs=output_layer)

```

Figure 7-57. Code to define the decoding stage and then the final layer. The model is then initialized

Now that the model has been defined, let's compile it and train it (see Figure 7-58).

```
TCN.compile(loss=keras.losses.categorical_crossentropy,
             optimizer=optimizers.Adam(lr=0.002),
             metrics=["accuracy"])

checkpointer = ModelCheckpoint(filepath="model_ED-
TCN_creditcard.h5",
                               verbose=0,
                               save_best_only=True)

TCN.summary()
```

Figure 7-58. *Compiling the model, defining the checkpoint callback, and calling the summary function*

The output should look somewhat like Figure 7-59.

Layer (type)	Output Shape	Param #
input_18 (InputLayer)	(None, 31, 1)	0
conv1d_101 (Conv1D)	(None, 31, 31)	93
max_pooling1d_35 (MaxPooling)	(None, 15, 31)	0
conv1d_102 (Conv1D)	(None, 15, 15)	945
max_pooling1d_36 (MaxPooling)	(None, 5, 15)	0
conv1d_103 (Conv1D)	(None, 5, 10)	310
dense_32 (Dense)	(None, 5, 5)	55
up_sampling1d_45 (UpSampling)	(None, 15, 5)	0
conv1d_104 (Conv1D)	(None, 15, 10)	110
up_sampling1d_46 (UpSampling)	(None, 30, 10)	0
conv1d_105 (Conv1D)	(None, 30, 15)	315
zero_padding1d_7 (ZeroPaddin	(None, 31, 15)	0
conv1d_106 (Conv1D)	(None, 31, 31)	961
flatten_15 (Flatten)	(None, 961)	0
dense_33 (Dense)	(None, 2)	1924
Total params: 4,713		
Trainable params: 4,713		
Non-trainable params: 0		

Figure 7-59. The summary of the model. This can help you get an idea of how the encoding and decoding works by looking at the output shapes of each layer

Notice the addition of the zero padding layer. What this layer does is add a 0 to the data sequence in order to help the dimensions match. Because the original data had an odd number of columns, the number of dimensions in the output of the decoder stage did not match the dimensions of the original data after being upsampled (this is because of rounding issues, since everything is an integer). To counter this,

```
zero_pad_1 = ZeroPadding1D(padding=(0,1))(conv_5)
```

was included, where the tuple is formatted as (left_pad, right_pad) to customize how the padding should be. Otherwise, passing in an integer will just pad on both ends. To summarize, **zero padding** will add a zero to each entry in the data to the left, right, or both (default) sides.

With the model compiled, all that's left for you to do is train the data (see Figure 7-60).

```
TCN.fit(x_train, y_train,
        batch_size=128,
        epochs=25,
        verbose=1,
        validation_data=(x_test, y_test),
        callbacks = [checkpointer])
```

Figure 7-60. Training the data on the training sets

After a while, you should end with something like Figure 7-61.

```
Epoch 7/25
6295/6295 [=====] - 0s - loss: 0.0891 - acc: 0.9889 - val_loss: 0.0909 - val_acc: 0.9907
Epoch 8/25
6295/6295 [=====] - 0s - loss: 0.0848 - acc: 0.9895 - val_loss: 0.0843 - val_acc: 0.9902
Epoch 9/25
6295/6295 [=====] - 1s - loss: 0.0804 - acc: 0.9900 - val_loss: 0.0827 - val_acc: 0.9907
Epoch 10/25
6295/6295 [=====] - 0s - loss: 0.0775 - acc: 0.9908 - val_loss: 0.0806 - val_acc: 0.9907
Epoch 11/25
6295/6295 [=====] - 0s - loss: 0.0749 - acc: 0.9911 - val_loss: 0.0811 - val_acc: 0.9905
Epoch 12/25
6295/6295 [=====] - 1s - loss: 0.0764 - acc: 0.9913 - val_loss: 0.0781 - val_acc: 0.9907
Epoch 13/25
6295/6295 [=====] - 0s - loss: 0.0752 - acc: 0.9903 - val_loss: 0.0788 - val_acc: 0.9907
Epoch 14/25
6295/6295 [=====] - 1s - loss: 0.0746 - acc: 0.9911 - val_loss: 0.0768 - val_acc: 0.9907
Epoch 15/25
6295/6295 [=====] - 1s - loss: 0.0733 - acc: 0.9916 - val_loss: 0.0826 - val_acc: 0.9905
Epoch 16/25
6295/6295 [=====] - 1s - loss: 0.0703 - acc: 0.9906 - val_loss: 0.0753 - val_acc: 0.9907
Epoch 17/25
6295/6295 [=====] - 0s - loss: 0.0731 - acc: 0.9906 - val_loss: 0.0741 - val_acc: 0.9909
Epoch 18/25
6295/6295 [=====] - 1s - loss: 0.0688 - acc: 0.9909 - val_loss: 0.0769 - val_acc: 0.9902
Epoch 19/25
6295/6295 [=====] - 1s - loss: 0.0695 - acc: 0.9916 - val_loss: 0.0754 - val_acc: 0.9909
Epoch 20/25
6295/6295 [=====] - 0s - loss: 0.0694 - acc: 0.9905 - val_loss: 0.0782 - val_acc: 0.9902
Epoch 21/25
6295/6295 [=====] - 0s - loss: 0.0696 - acc: 0.9909 - val_loss: 0.0765 - val_acc: 0.9905
Epoch 22/25
6295/6295 [=====] - 1s - loss: 0.0723 - acc: 0.9898 - val_loss: 0.0739 - val_acc: 0.9905
Epoch 23/25
6295/6295 [=====] - 1s - loss: 0.0687 - acc: 0.9909 - val_loss: 0.0710 - val_acc: 0.9912
Epoch 24/25
6295/6295 [=====] - 1s - loss: 0.0665 - acc: 0.9913 - val_loss: 0.0703 - val_acc: 0.99140.
Epoch 25/25
6295/6295 [=====] - 0s - loss: 0.0657 - acc: 0.9916 - val_loss: 0.0718 - val_acc: 0.9905

Out[73]: <keras.callbacks.History at 0x23907fc6be0>
```

Figure 7-61. This output is similar to what you should see after the training process ends

Now evaluate your model's performance (see Figure 7-62).

```
score = TCN.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Figure 7-62. *Evaluates the model's performance in terms of loss and accuracy*

You should see an output similar to Figure 7-63.

```
In [74]: 1 score = TCN.evaluate(x_test, y_test, verbose=0)
          2 print('Test loss:', score[0])
          3 print('Test accuracy:', score[1])

Test loss: 0.0717651191317761
Test accuracy: 0.9904693828925423
```

Figure 7-63. *The generated outputs for loss and accuracy for the model when the test sets are passed in*

Pretty good, but how's the AUC score? Run the code in Figure 7-64.

```
from sklearn.metrics import roc_auc_score

preds = TCN.predict(x_test)
auc = roc_auc_score( np.round(preds), y_test)
print("AUC: {:.2%}".format (auc))
```

Figure 7-64. *Code to check the AUC score given the rounded predictions and the test sets*

The output should look somewhat like Figure 7-65.

```
In [75]: 1 from sklearn.metrics import roc_auc_score
          2
          3 preds = TCN.predict(x_test)
          4 auc = roc_auc_score( np.round(preds), y_test)
          5 print("AUC: {:.2%}".format (auc))

AUC: 98.64%
```

Figure 7-65. *The generated AUC score*

That’s a nice AUC score! So for both the encoder-decoder TCN and dilated TCN architectures, you’ve managed to attain AUC scores of over 98% on the credit card data set in a supervised setting. Although both models trained and performed in a supervised setting, since the anomalies and the normal entries were labeled as such, the key takeaway is that TCNs are incredibly quick to train with GPUs and can perform really well.

Summary

In this chapter, we discussed temporal convolutional networks and showed how they fare when applied to anomaly detection.

In the next chapter, we will look at practical use case of anomaly detection.

CHAPTER 8

Practical Use Cases of Anomaly Detection

In this chapter, you will learn how anomaly detection can be used in several industry verticals. You will explore how anomaly detection techniques can be used to address practical use cases and address real-life problems in the business landscape. Every business and use case is different, so while we cannot copy-paste code to build a successful model to detect anomalies in any dataset, this chapter will cover many use cases to give an idea of the possibilities and concepts behind the thought processes.

In a nutshell, the following topics will be covered throughout this chapter:

- What is anomaly detection?
- Real-world use cases of anomaly detection
 - Telecom
 - Banking
 - Environmental
 - Healthcare
 - Transportation
 - Social Media
 - Finance and Insurance
 - Cybersecurity
 - Video Surveillance
 - Manufacturing

- Smart Homes
- Retail
- Implementation of deep learning-based anomaly detection

Anomaly Detection

Anomaly detection is finding patterns that do not adhere to what is considered as normal or expected behavior. Businesses can lose millions of dollars due to abnormal events. Consumers can also lose millions of dollars. In fact, there are many situations every day where people's lives are at risk and where their property is at risk. If your bank account gets cleaned out, that's a problem. If your water line breaks, flooding your basement, that's a problem. If all flights get delayed, that's a problem. You might have been misdiagnosed or not diagnosed at all with a health issue, which is a very big problem that directly impacts your well-being.

Figure 8-1 is an example of an anomaly showing a rainbow-colored fish in the blueish fish family.

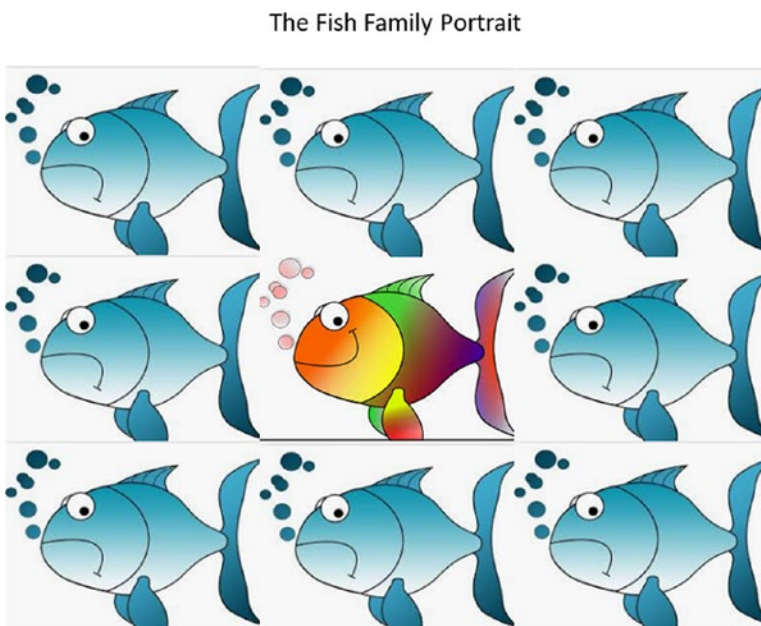


Figure 8-1. *An example of an anomaly*

In business use cases, everything is centered around data, and anomaly detection is the identification of abnormal data points, events, or observations that raise suspicions due to the fact that they differ significantly from the data perceived as normal or typical. Many such anomalies can impact the business operations or bottom lines significantly, which is why anomaly detection is gaining a lot of traction in certain industries and many businesses are investing heavily in technologies that can help them identify abnormal behavior before it is too late. Such proactive anomaly detection is becoming more and more visible, and due to the new technologies developed as part of the AI revolution, this problem is also getting solved in ways never possible before.

Figure 8-2 is an example of the daily number of cars that cross the Golden Gate Bridge in San Francisco.

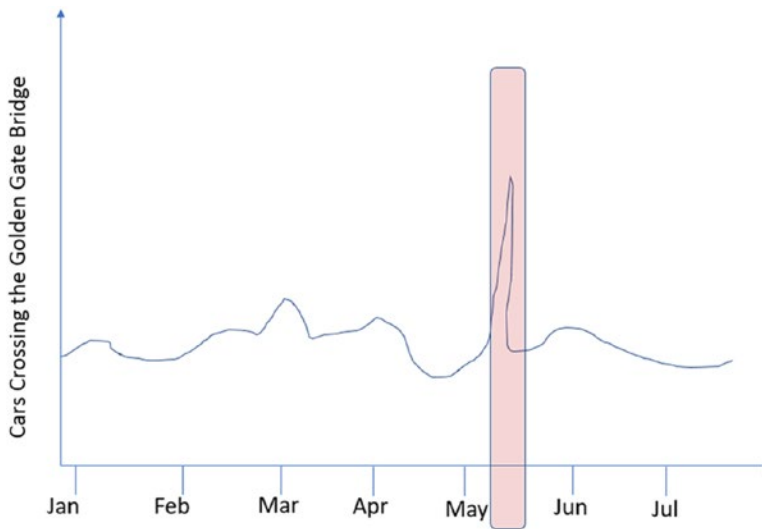


Figure 8-2. *Daily count of cars crossing*

The kind of anomaly detection that can potentially help businesses depends very much on the kind of data collected as part of the business operations and the kind of techniques and algorithms used as part of the strategy to perform the anomaly detection.

Real-World Use Cases of Anomaly Detection

We will look at several industry verticals and businesses, and how anomaly detection can be used.

Telecom

In the telecom sector, some of the use cases for anomaly detection are to detect roaming abuse, revenue fraud, and service disruptions. So how do we detect roaming abuse in the telecom sector? By looking at the location of the cellular devices, we can categorize the kind of behavior of the cellular device at any particular moment as normal or abnormal. This helps us detect cellular device usage at that period of time. By looking at all of the other information we know in general about roaming activity, we can also detect how this cellular device is being used and whether any roaming abuse is taking place. Figure 8-3 shows how roaming works for your phone as you travel around the world.



Figure 8-3. *Roaming*

Service disruption is another very high impact use case for anomaly detection. Cellular devices are connected to cellular networks via towers, which are all over the place. Your cell phone connects to the nearest tower in order to participate in the cellular network. In case of events involving large crowds such as a concert or a football game, the cellular towers that typically perform quite well get heavily overloaded, causing serious service disruptions and very bad customer experience for the duration of the overload. Figure 8-4 shows a service disruption of phone service in the northwestern United States.

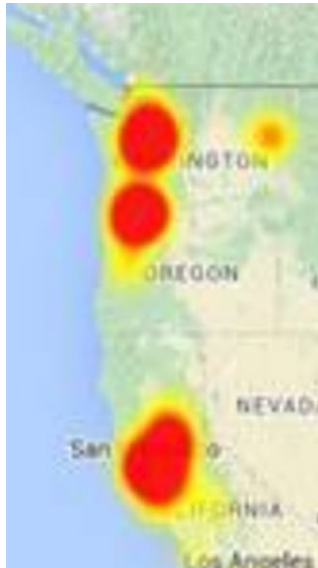


Figure 8-4. *Service disruptions*

If we know the various metrics of the cell phone towers and the associated devices at some period of time and for a long duration, along with any kind of information we have on the typical nature of activity around the towers in terms of whether there were concerts or games in the vicinity or a major event is expected in the vicinity of the cellular towers, we can use a time series as a basis to represent all such activity and subsequently use TCN or LSTM algorithms to detect anomalies pertaining to the major events because they have a temporal dependency. This will help in looking at how these services are being used and how effective the service is for the particular cell phone towers.

The cell phone companies now have a way of understanding whether certain hours need to be upgraded or more towers need to be built. For instance, if major office buildings are being built near a particular tower, using data on the time series of all the towers owned by the cellular network, it is possible to detect anomalies in other parts of the network and apply the principles to the tower that is probably going to be impacted by the newly constructed office buildings (which will add thousands of cell phone connections and could cause overloading on the tower and affect how the tower will be used in the near future).

Banking

In the banking sector, some of the use cases for anomaly detection are to flag abnormally high transactions, fraudulent activity, phishing attacks, etc. Credit cards are used by almost everyone in the world, and typically every individual has a certain way of using their credit card, which is different from everyone else. So there is an implicit profile of the individual using the credit card in terms of how they use it, when they use it, why they use it, and what did they use it for. If the credit card company has such information about the credit card usage of very large number of consumers, it is possible to use anomaly detection to detect when a specific credit card transaction may be fraudulent.

Autoencoders are very useful in such an anomaly detection use case. With such a case, we can take all the credit card transactions by individual consumers, and capture and convert the features into numerical features such that we can assign certain scores to every credit card based on various factors along with a kind of indicator as to whether the transaction are normal or abnormal. Then, using autoencoders, we can build an anomaly detection model that can quickly determine a specific transaction as normal or abnormal given everything we know about all the other transactions for a customer. The autoencoder does not even need to be extremely complicated; it can be built with just a few hidden layers for the encoder and a few hidden layers for the decoder and still have pretty decent detection of abnormal activity (otherwise known as fraudulent activity) on the credit cards. Figure 8-5 is a depiction of credit card fraud.

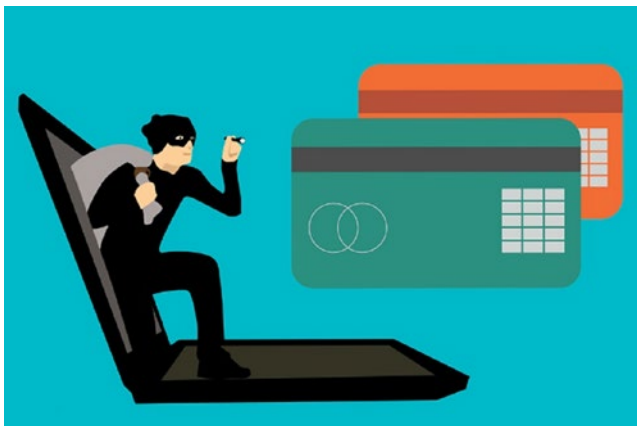


Figure 8-5. *Depiction of credit card fraud*

Environmental

When it comes to environmental aspects, anomaly detection has several applicable use cases. Whether it is deforestation or melting of glaciers, air quality or water quality, anomaly detection can help in identifying abnormal activities. Figure 8-6 is a photo of deforestation.



Figure 8-6. *Deforestation*
Source: commons.wikimedia.org

Let's look at an example of the air quality index. The air quality index provides some kind of measurement of breathable air quality, which can be measured by using various sensors placed at various locations in the region. These sensors measure and send periodic data to be collected by a centralized system where such data is collected from all of the sensors. This becomes a time series, with each measurement consisting of several attributes or features. With each point in time having a certain number of features, which can then be input into a neural network such as an autoencoder, we can build an anomaly detector. Of course, we can use a LSTM or even TCN to do the same. Figure 8-7 shows the air quality index in Seoul in 2015.

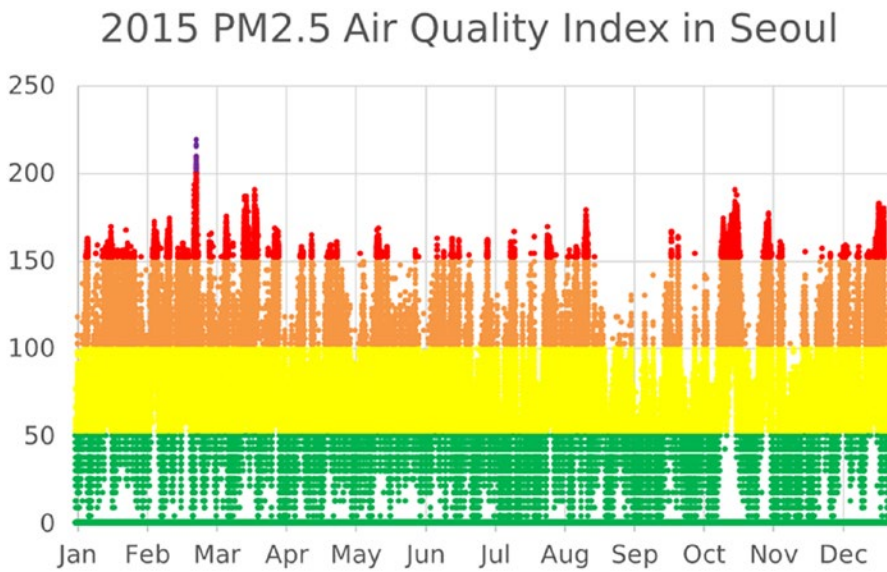


Figure 8-7. Air quality index

Source: commons.wikimedia.org

Healthcare

Healthcare is one of the domains that can benefit a lot from anomaly detection, whether it is to prevent fraud, detect cancer or chronic illness, improve ambulatory services, etc.

One of the biggest use cases for anomaly detection in healthcare is to detect cancer from various diagnostic reports even before there are any significant symptoms that might indicate the presence of cancer. This is extremely important given the serious consequences of cancer for any person. Some of the techniques in anomaly detection that we can use here involve convolutional neural networks combined with autoencoders.

Convolutional neural networks use the concept of dimensionality reduction to reduce the large number of features/pixels with colors into much lower dimensionality points using the neural networks layers. So, if we combine this convolutional neural network with autoencoders, we can also use autoencoders to look at images such as MRI images, mammograms, or other images from diagnostic technologies in the healthcare industry. Figure 8-8 is a set of images from a CT scan.

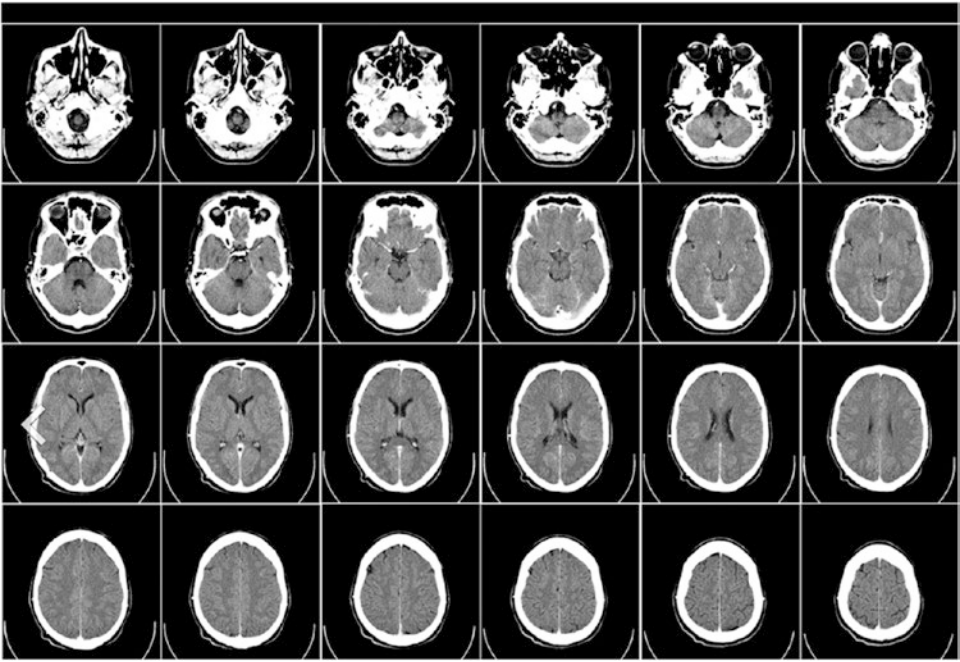


Figure 8-8. CT scan images
Source: commons.wikimedia.org

Let's look at another use case of detecting abnormal health conditions of residents of a particular neighborhood. Typically, local hospitals are used by residents of specific neighborhoods. Using such data, the hospital can collect and store various kinds of health metrics from all the residents in this neighborhood. Some of the possible metrics are blood test results, lipid profiles, glycemic values, blood pressure, ECG, etc. When combined with demographic data such as age, sex, health conditions, etc., this information potentially allows us to build a sophisticated AI-based anomaly detection model.

Figure 8-9 shows different health issues observed by looking at ECG results.

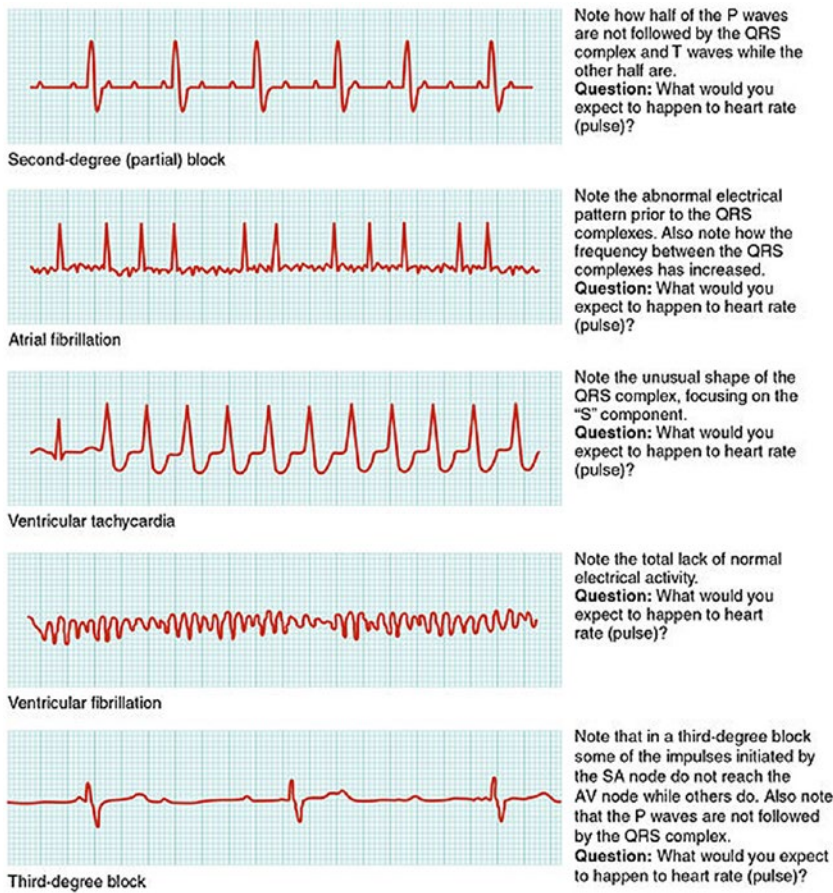


Figure 8-9. ECG results
Source: commons.wikimedia.org

There are a lot of different use cases in healthcare where we can use different anomaly detection algorithms to implement preventative measures.

Transportation

In the transportation sector, anomaly detection can be used to ensure proper functioning of the roadways and vehicles. If we can collect different types of events from all the sensors that are operational on the roadways such as toll booths, traffic lights, security cameras, and GPS signals, we can build an anomaly detection engine that we can then use to detect abnormal traffic patterns.

Anomaly detection can also be used to look at times in schedules of public transportation and the related traffic conditions in the similar area of transportation.

We can also look for abnormal activity in terms of fuel consumption, number of passengers the public transportation is supporting, seasonal trends, etc. Figure 8-10 is an image of a traffic jam due to peak time unexpected traffic.



Figure 8-10. *Traffic jam*

Social Media

In social media platforms such as Twitter, Facebook, and Instagram, anomaly detection can be used to detect hacked accounts spamming everyone, false advertisements, fake reviews, etc. Social media platforms are used extensively by billions of people, so the amount of activity on social media platforms is extremely high and is ever growing. In order to ensure the privacy of the individuals using the social media platforms as well as to ensure the proper experience for each and every individual using the social media platforms, there are many techniques that can be used to enhance the capabilities of this system. Using anomaly detection, every individual activity can be examined for normal and abnormal behavior.

Similarly, any advertising platforms ads, any personalized friend recommendations, any news articles that the individual might have been interested in, such as elections,

can be processed for abnormal or anomalous activity. It would be a great use case for anomaly detection if anomaly detection could detect troll activity on your tweets, propagandized bots, fake news, and so on. Anomaly detection can also be used to detect if your account has been taken over, because all of a sudden your account might be posting an immense amount of tweets, pause tweets, and comments, or might be trolling other accounts and spamming everyone else. Figure 8-11 shows an article on fake news on Facebook.



Figure 8-11. Fake news on Facebook

Finance and Insurance

In the finance and insurance industries, anomaly detection can be used to detect fraudulent claims, fraudulent transactions such as transfer of money in and out of the country, fraudulent travel expenses, and the risk associated with the specific policy or individual, etc. The finance and insurance industries depend on the ability to target the right consumers and take the right amount of risk when dealing with finance and

insurance. For instance, if they already know that a specific area is prone to forest fires or earthquakes or very frequent flooding, the insurance company insuring your home needs to have all the tools that they can get their hands on to quantify the amount of risk involved when writing the policy for homeowner insurance.

Anomaly detection can also be used to detect wire fraud where a large amount of money is transferred in and out of the country using several different accounts, something extremely difficult for human eyes to manually glance over and figure out considering the massive volume of transactions that can take place every hour. This is feasible because AI techniques can be trained on very large amounts of data to detect very new and innovative wire fraud beyond the capabilities of any human or many of the statistical techniques that have been in place for decades. Deep learning does solve a very big problem in the financial and insurance industries, and with the advent of graphical processing units (GPUs), this is becoming a reality in many of the very hard-to-crack use cases. Anomaly detection and deep learning can be used together in order to serve the needs of the business. Figure 8-12 shows the mortgage loan fraud reporting trend.

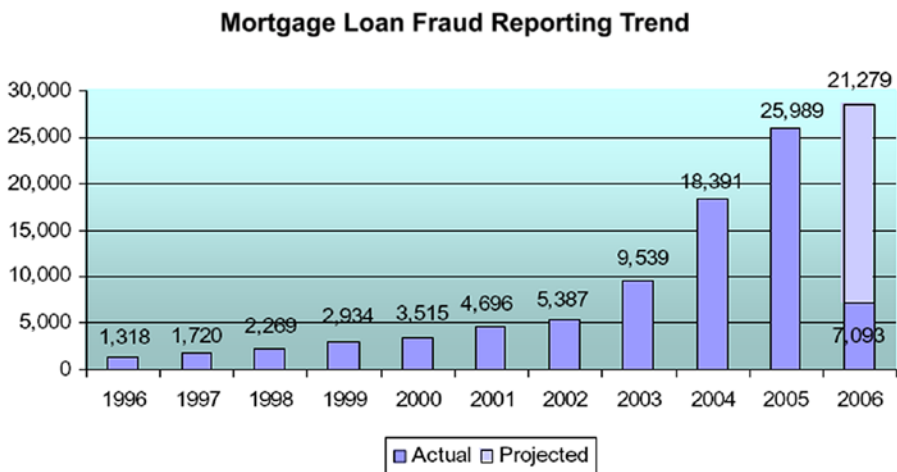


Figure 8-12. Mortgage loan fraud reporting trend

Cybersecurity

Another use case for anomaly detection is in cybersecurity or networking. In fact, one of the very first use cases for anomaly detection was decades ago when just the statistical models were being used to try to detect any intrusion attempts into networks. In the cybersecurity space, there are many things that can happen. One of the most prevalent

attacks is a denial of service (DOS) attack. When a denial of service attack is launched against your company’s website or portal so as to disrupt service to your customers, typically a large number of machines are mobilized to run simultaneous connections and random useless transactions against your portal (which is probably dealing with some kind of a payment service for customers). As a result, the portal isn’t responsive to the customers, eventually leading to very poor customer experience and a loss of business.

Anomaly detection can detect the anomalous activity since we’re training the system on data that has been collected for a long period of time. This data is comprised of typical use behavior, patterns in payment, how many users are active, and how much the payment is at this particular time, as well as seasonal behaviors and other trends that exist for the payment portal. When a DOS attack is suddenly launched against your payment portal, it is very possible for your anomaly detection algorithm to detect such activity and quickly notify the infrastructure or operational teams who can take corrective action such as setting up different firewall rules or better routing rules that attempt to block the anomalous or bad actors from launching the attack or prolonging the attack against the portal. Figure 8-13 is example of anomaly monitoring network flows.

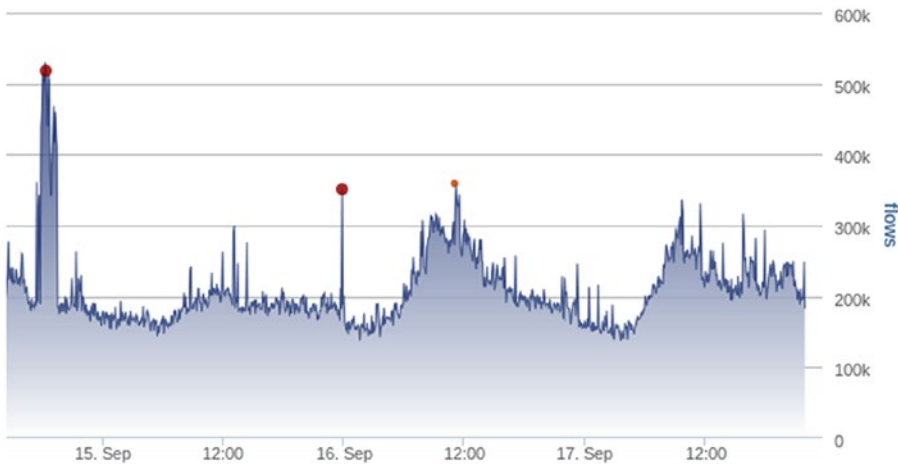


Figure 8-13. *Anomaly monitoring network flows*

Another example is when hackers try to get into a system given that they were somehow able to set up a Trojan to get into the network in the first place. Typically, this process involves a lot of scanning, such as port or IP scanning, to see what machines exist in the network when the services are being run. The machines may be running SSH and telnet (which is easier to crack), and the hacker may try to launch several different types

of attacks that exploit the vulnerabilities of the telnet or asset service. Eventually, one of the targeted machines will respond, and the hacker will get into the system and continue the penetration of the internal network until they accomplish what they came for.

Typically, networks have a pattern of usage, and there are database servers, web servers, development servers, payroll systems, QA systems, and end user-facing systems. Usually the well-known, expected behavior is seen for a long period of time. Then there is a change that is observed and expected over a long period of time as to how the machines are used as well as how the networks are used. We can also measure the ways machines talk to each other and via which service/ports.

Using anomaly detection, we can detect if a specific port or service on a specific machine or machines is being connected to or transacted with at an abnormal rate, meaning that there is some kind of intrusion activity taking place where some intruder is trying to hack into the specific system or systems. This is extremely valuable information to the operations team, who can quickly pull in the cybersecurity experts and try to drill down into what is really going on and take any kind of preventive or proactive action rather than reactivate. This could be the difference between the business staying afloat or the business shutting down (at least temporarily). There have been instances where a single cyber security intrusion almost bankrupted a business, costing hundreds of millions of dollars in damages. This is the reason why the cybersecurity domain is very interested in deep learning, and the use cases that involve deep learning anomaly detection are some of the top use cases in the cyber security and networking space in this day and age. Figure 8-14 shows an anomaly in the number of TCP connections on different service ports.

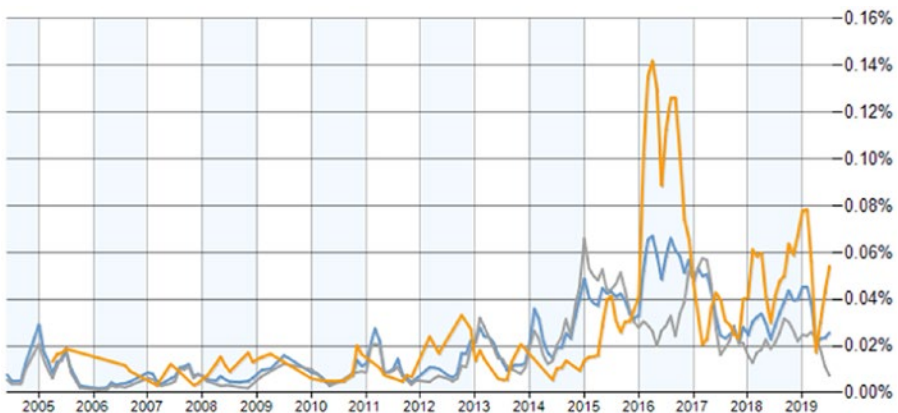


Figure 8-14. TCP connections over service ports

Not all the use cases are doom and gloom in cyber security or networking; anomaly detection can also be involved in determining whether we need to upgrade some of the systems, whether our systems are able to sustain the traffic for now and in the future, whether any node capacity planning needs to take place to bring everything back to normal, and so on. This is again very important for the operations team so it can understand if there are trends which were not foreseen a year ago that are now affecting the normal to abnormal behavior of the network. It is very important to know right now rather than later when it is too late and to start proactively planning to deal with this origin traffic or transactions that are happening in our network against some specific machine or machines.

Video Surveillance

Another domain where anomaly detection is becoming extremely important is video surveillance. Nowadays, it is very common to see security cameras and video surveillance systems no matter where you go: a local school, a local park, Main Street, near a neighbor's house, or in your own house. The point is, video surveillance is here to stay. Given all the new technological advancements in smart apps and smartphones, this is definitely not going to change any time soon. Rather, we should expect much more video surveillance. In the very near future, we will see lot more smart cars and self-driving cars. They also depend on continuous processing of video using real-time analysis and detecting various objects. At the same time, they can also detect any kind of anomaly. In a strictly security video surveillance sense, anomaly detection can be used to detect the normal for the specific camera that is looking at your backyard. When a specific anomaly is detected because of some kind of motion within the vicinity of your house, such as a wild animal or even an intruder walking on your lawn, your home security system is able to see that this is not normal. In order for the cameras to do this effectively, the manufacturers train very sophisticated machine learning models to assess the video signals in real time. The feed coming from the cameras is determined as normal or abnormal. For example, if you are driving in a self-driving car on the interstate, video of the car will clearly indicate what is normal right now according to how the road should look, where the signs should be, where the trees should be, and where the next car should be. Using anomaly detection, self-driving cars can avoid any abnormalities happening on the path and then take corrective action before anything bad can happen.

Figure 8-15 is an object-detecting video surveillance system.



Figure 8-15. Object-detecting video surveillance system

Manufacturing

Anomaly detection is also being used heavily in the manufacturing sector. Specifically, since most of the manufacturing nowadays involves robots and a lot of automation, anomaly detection can be used to detect malfunctions or impending failure of parts of the manufacturing system.

In the manufacturing industry, because of all the automation that is happening, there is a lot of emphasis on various kinds of sensors and other types of metrics being collected in a real-time or near real-time basis. This data can be used to build a sophisticated anomaly detection model to try to detect if there is any impending problem that will be seen very soon in the plant or the manufacturing cycle.

Another example of anomaly detection and how it can be used in business is the case of oil and natural gas platforms. An oil and natural gas platform typically has thousands of components all interconnected in various ways in order to make the plant functional. Needless to say, all the components can be monitored using sensors that do specific measurements of the various parameters of the components to which the sensors are attached to. All these sensors can be part of an IoT (Internet of Things) platform. If you

can collect all the sensor output from the tens of thousands of sensors attached to the tens of thousands of components, then it becomes possible for us to collect such data for a longer period of time and train sophisticated anomaly detection models such as autoencoders, LSTMs, and TCNs.

Figure 8-16 shows a manufacturing plant with sensor readings.

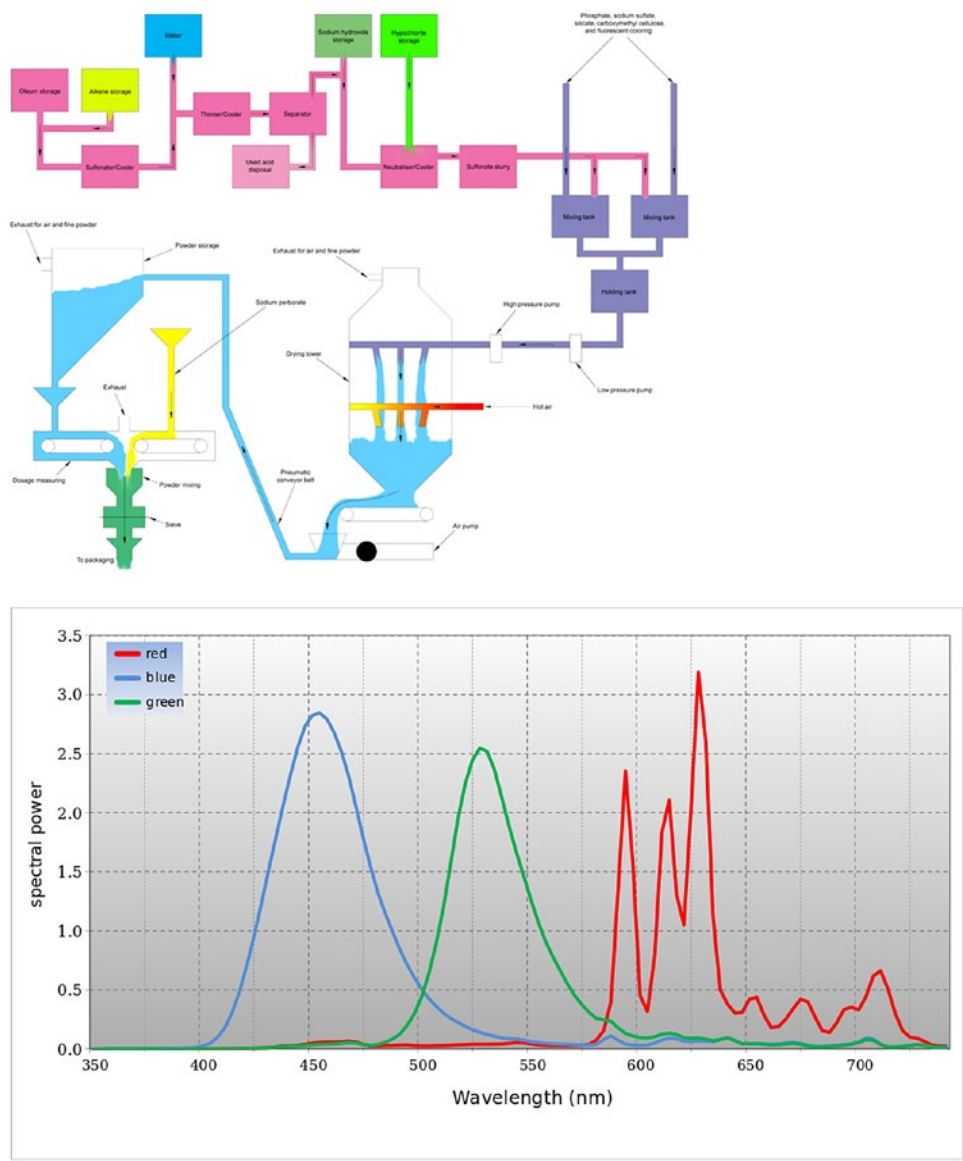


Figure 8-16. Manufacturing plant with sensor readings

Smart Home

Another kind of business that is also using anomaly detection to its advantage is the smart home system. Smart homes have lots of integrated components, such as smart thermostats, refrigerators, and interconnected devices, that all talk to each other. Let's say you have an Amazon Alexa. Alexa can talk to your smart lights, which use smart bulbs. All components can use a very smart app on your smart phone. Even thermostats are interconnected. So how do we really use anomaly detection in this use case? A simple way is to monitor how you set your thermostat for the optimal temperature during all weather conditions and follow some sort of recommendation or recommended behavior. Because the thermostats are personalized to some extent in each household, there may be a very good deep learning algorithm out there that is continuously looking for the thermostats across all houses, including yours, and can then detect how you use it normally. Figure 8-17 is an illustration of a smart home.



Figure 8-17. *A smart home*

Retail

Another big industry that uses anomaly detection algorithms is the retail industry. In the retail industry, there are certain use cases such as the efficiency of the supply chain in terms of distribution of goods and services. Also interesting are the returns from customers because returned goods are tricky: sometimes it costs less to sell them in a clearance sale than to restock.

Looking at customer sales is also critical both in terms of revenue generated by sales and in terms of planning future products and sales strategies, especially when it comes to targeting the consumers better. Figure 8-18 shows the historical sales figures of a product.

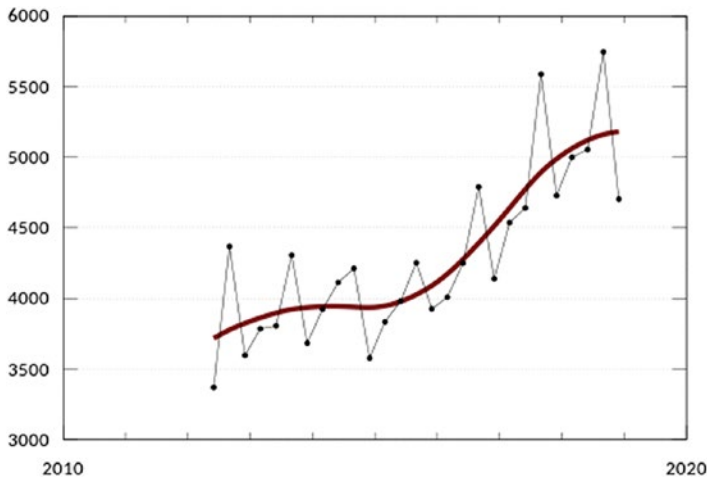


Figure 8-18. *Historical sales figures of a product*

Implementation of Deep Learning-Based Anomaly Detection

Given these use cases in these different industries, what are the key steps in establishing an anomaly detection practice in your organization or business?

The key steps involved in anomaly detection are as follows:

- Identifying business use case and getting aligned on the expectations
- Defining what data is available and understanding it and the nature of the data itself
- Establishing the processes to consume the data in order to process it
- Establishing the type of models to use
- A strategic discussion of how the models will be used and executed

- Investigating the results and feedback analysis as it effects the business
- Operationalizing the model used in the day-to-day activity of the business

In particular, we are very interested in how the models are built and in what type of models we should be using. The type of anomaly detection algorithm used affects pretty much everything that we are trying to get out of this anomaly detection strategy. This in turn depends on the type of data available, as well as whether the data is already labeled or identified. One of the things that will affect the decision to figure out what type of anomaly detection will work best for the specific use case is whether it is a point anomaly, contextual anomaly, or a collective anomaly. We are also interested in looking at whether the data is an instantaneous snapshot at some point in time or if it is continuously evolving or ever-changing, real-time, time series data. Also important is whether the specific features or attributes of the data are categorical or numerical, nominal, ordinal, binary, discrete, or continuous. It is also very important to know if the data is being labeled already or if some sort of a hint is provided as to what this data is, since it could steer us in the direction of supervised, semi-supervised, or unsupervised algorithms.

While the technologies and algorithms are available to be used, there are several key challenges to implementing an anomaly detection approach based on deep learning:

- It's hard to integrate AI into existing processes and systems.
- The technologies and the expertise needed are expensive.
- Leadership needs to be educated on what AI can and cannot do.
- AI algorithms are not natively intelligent; rather, they learn by analyzing "good" data.
- There is a need for change in "culture," especially in large companies.

Summary

In this chapter, we discussed practical use cases of anomaly detection in the business landscape. We showed how anomaly detection can be used to address real-life problems in many businesses. Every business and use case is different, so while we cannot copy/

paste code to build a successful model to detect anomalies in any dataset, this chapter covered many use cases to give you an idea of the possibilities and concepts behind the thought process.

Remember that this is an evolving field with continuous inventions and enhancements to the algorithms present, which means that in the future the algorithms will not look the same. Just couple of years ago, the RNN (recurrent neural network) was the best algorithm for a time series, but now the LSTM (Chapter 6) is being used heavily and the TCN (Chapter 7) will be the future of dealing with a time series. Even autoencoders have changed quite a bit; the traditional autoencoders have evolved into variational autoencoders (Chapter 4). The RBM (Chapter 5) is not used that much any longer.

In the next chapter, Appendix A, we will look at Keras, which is a popular framework for deep learning.

APPENDIX A

Intro to Keras

In this appendix, you will be introduced to the Keras framework along with the functionality that it offers. You will also take a look at using the back end, which is TensorFlow in this case, to perform low-level operations all using Keras.

Regarding the setup, we use

- tensorflow-gpu version 1.10.0
- keras version 2.0.8
- torch version 0.4.1 (this is PyTorch)
- CUDA version 9.0.176
- cuDNN version 7.3.0.29

What Is Keras?

Keras is a high-level, deep learning library for Python, running with TensorFlow, CNTK, or Theanos as the **back end**. The back end can basically be thought of as the “engine” that does all of the work, and Keras is the rest of the car, including the software that interfaces with the engine.

In other words, Keras being high-level means that it abstracts away much of the intricacies of a framework like TensorFlow. You only need to write a few lines of code to have a deep learning model ready to train and ready to use. In contrast, TensorFlow being more of a low-level framework means you have much more added syntax and functionality to define the extra work that Keras abstracts away for you. At the same time, TensorFlow and PyTorch also allow for much more flexibility if you know what you’re doing.

TensorFlow and PyTorch allow you to manipulate individual **tensors** (similar to matrices, but they aren't limited to two dimensions; they can range from vectors to matrices to n-dimensional objects) to create custom neural network layers, and to create new neural network architectures that include custom layers.

With that being said, Keras allows you to do the same things as TensorFlow and PyTorch do, but you will have to import the back end itself (which in this case is TensorFlow) to perform any of the low-level operations. This is basically the same thing as working with TensorFlow itself since you're using the TensorFlow syntax through Keras, so you still need to be knowledgeable about TensorFlow syntax and functionality.

In the end, if you're not doing research work that requires you to create a new type of model, or to manipulate the tensors directly, simply use Keras. It's a much easier framework to use for beginners, and it will go a long way until you become sufficiently advanced enough that you need the low-level functionality that TensorFlow or PyTorch offers. And even then, you can still use TensorFlow (or whatever back end you're using) through Keras if you need to do any low-level work. One thing to note is that Keras has actually been integrated into TensorFlow, so you can access Keras through TensorFlow itself, but for the purpose of this appendix, we will use the Keras API to showcase the Keras functionality, and the TensorFlow back end through Keras to demonstrate the low-level operations that are analogous to PyTorch.

Using Keras

When using Keras, you will most likely import the necessary packages, load the data, process it, and then pass it into the model. In this section, we will cover model creation in Keras, the different layers available, several submodules of Keras, and how to use the back end to perform tensor operations.

If you'd like to learn Keras even more in depth, feel free to check out the official documentation. We only cover the basic essentials that you need to know about Keras, so if you have further questions or would like to learn more, we recommend you to explore the documentation.

For details on implementation, Keras is available on GitHub at <https://github.com/keras-team/keras/tree/c2e36f369b411ad1d0a40ac096fe35f73b9dffd3>.

The official documentation is available at <https://keras.io/>.

Model Creation

In Keras, you can build a **sequential model**, or a **functional model**.

The **sequential model** is built as shown in Figure A-1.

```
In [2]: 1  ### Sequential model
        2
        3  import keras
        4  from keras.models import Sequential
        5  from keras.layers import Dense
        6
        7  seq_model = Sequential()
        8  seq_model.add(Dense(16, input_shape=(8,)))
        9  seq_model.add(Dense(32, activation='relu'))
       10  seq_model.add(Dense(16, activation='softmax'))
       11
```

Figure A-1. Code defining a sequential model in Keras

Once you’ve defined a sequential model, you can simply add layers to it by calling `model_name.add()`, where the layer itself is the parameter. Once you’ve finished adding all of the layers that you want, you are ready to compile and train the model on whatever data you have.

Now, let’s look at the **functional model**, the format of which is what you’ve used in the book thus far (see Figure A-2).

```
In [5]: 1  ### Functional model
        2
        3  import keras
        4  from keras.models import Model
        5  from keras.layers import Input, Dense
        6
        7  input_layer = Input(shape=(8,))
        8  dense_1 = Dense(32, activation='relu')(input_layer)
        9  output_layer = Dense(16, activation='softmax')(dense_1)
       10
       11  func_model = Model(input_layer, output_layer)
```

Figure A-2. Code defining a functional model in Keras

The functional model allows you to have more flexibility in how you define your neural network. With it, you can connect layers to any other layer that you want, instead of being limited to just the previous layer like in the sequential model. This allows you to share a layer with multiple other layers or even reuse the same layer, allowing you to create more complicated models.

Once you're done defining all of your layers, you simply need to call `Model()` with your input and output parameters respectively to finish your whole model. Now, you can continue onwards to compiling and training your model.

Model Compilation and Training

In most cases, the code to compile your model will look something like Figure A-3.

```
In [ ]: 1 model.compile(optimizer="",
2                       loss="",
3                       metrics="")
```

Figure A-3. Code to compile a model in Keras

However, there are many more parameters to consider:

- **optimizer:** Passes in the name of the optimizer in the string or an instance of the optimizer (you call the optimizer with whatever parameters you like. We will elaborate on this further below in the **Optimizers** section.)
- **loss:** Passes in the name of the loss function or the function itself. We elaborate on what we mean by this below in the **Losses** section.
- **metrics:** Passes in the list of metrics that you want the model to evaluate during the training and testing processes. Check out the **Metrics** section for more details on what metrics you can use.
- **loss_weights:** If you have multiple outputs and multiple losses, the model evaluates based on the total loss. The `loss_weights` are a list or dictionary that determines how much each loss factors into the overall, combined loss. With the new weights, the overall loss is now the weighted sum of all losses.
- **sample_weight_mode:** If your data has 2D weights with timestep-wise sample weighting, then you should pass in "temporal". Otherwise, None defaults to 1D sample-wise weights. You can also pass a list or dictionary of `sample_weight_modes` if your model has multiple outputs. One thing to note is that you need at least a 3D output, with one dimension being time.

- **weighted_metrics:** A list of metrics for the model to evaluate and weight using `sample_weight` or `class_weight` during the training and testing processes.

After compiling the model, you can also call a function to **save** your model as in Figure A-4.

```
In [ ]: 1 from keras.callbacks import ModelCheckpoint
        2
        3 checkpointer = ModelCheckpoint(filepath="saved_model.h5",
        4                               verbose=0,
        5                               save_best_only=True)
```

Figure A-4. A callback to save the model to some file path

Here are the set of parameters associated with `ModelCheckpoint()`:

- **filepath:** The path where you want to save the model file. Typing just “saved_model.h5” saves it in the same directory.
- **monitor:** The quantity that you want the model to monitor. By default, it’s set to “val_loss”.
- **verbose:** Sets verbosity to 0 or 1. It’s set to 0 by default.
- **save_best_only:** If set to True, then the model with the best performance according to the quantity monitored will be saved.
- **save_weights_only:** If set to True, then only the weights will be saved. Essentially, if True, `model.save_weights(filepath)`, else `model.save(filepath)`.
- **mode:** Can choose between auto, min, or max. If `save_best_only` is True, then you should pick a choice that would suit the monitored quantity best. If you chose `val_acc` for monitor, then you want to pick max for mode, and if you choose `val_loss` for monitor, pick min for mode.
- **period:** How many epochs there are between each checkpoint.

Now, you can train your model using code similar to Figure A-5.

```
In [ ]: 1 model.fit(x, y,
2             batch_size=128,
3             epochs=25,
4             verbose=1,
5             validation_data=(x_t, y_t),
6             callbacks = [checkpointer])
```

Figure A-5. Code to train the model

The `model.fit()` function has a big list of parameters:

- **x:** This is a Numpy array representing the training data. If you have multiple inputs, then this is a list of Numpy arrays that are all training data.
- **y:** This is a Numpy array that represents the target or label data. Again, if you have multiple outputs, then this is a list of target data Numpy arrays.
- **batch_size:** Set to 32 by default. This is the integer number of samples to run through the network before updating the gradients.
- **epochs:** An integer value dictating how many iterations for the entire x and y data to pass through the network.
- **verbose:** 0 makes it train without outputting anything, 1 shows a progress bar and the metrics, and 2 shows one line per epoch. Check the figures below for exactly what each value does:

Verbosity 1 (Figure A-6)

```
In [16]: 1 TCN.fit(x_train, y_train,
2             batch_size=128,
3             epochs=25,
4             verbose=1,
5             validation_data=(x_test, y_test),
6             callbacks = [checkpointer])
7
```

```
Train on 6295 samples, validate on 4197 samples
Epoch 1/25
6295/6295 [=====] - 0s - loss: 0.0620 - acc: 0.9911 - val_loss: 0.0641 - val_acc: 0.9900
Epoch 2/25
6295/6295 [=====] - 0s - loss: 0.0656 - acc: 0.9895 - val_loss: 0.0655 - val_acc: 0.9890
Epoch 3/25
6295/6295 [=====] - 0s - loss: 0.0622 - acc: 0.9905 - val_loss: 0.0630 - val_acc: 0.9907
Epoch 4/25
3712/6295 [=====>.....] - ETA: 0s - loss: 0.0637 - acc: 0.9903
```

Figure A-6. The training function with verbosity 1

Verbosity 2 (Figure A-7)

```
In [17]: 1 TCN.fit(x_train, y_train,
          2         batch_size=128,
          3         epochs=25,
          4         verbose=2,
          5         validation_data=(x_test, y_test),
          6         callbacks = [checkpointer])
          7

Train on 6295 samples, validate on 4197 samples
Epoch 1/25
1s - loss: 0.0633 - acc: 0.9900 - val_loss: 0.0639 - val_acc: 0.9914
Epoch 2/25
0s - loss: 0.0621 - acc: 0.9905 - val_loss: 0.0626 - val_acc: 0.9914
Epoch 3/25
0s - loss: 0.0639 - acc: 0.9897 - val_loss: 0.0637 - val_acc: 0.9912
Epoch 4/25
```

Figure A-7. *The training function with verbosity 2*

- **callbacks:** A list of `keras.callbacks.Callback` instances. Remember the `ModelCheckpoints` instance defined earlier as “`checkpointer`”? This is where you include it. To see how it’s done, refer to one of the above figures that showcase the `model.fit()` function being called.
- **validation_split:** A float value between 0 and 1 that tells the model how much of the training data should be used as validation data.
- **validation_data:** A tuple `(x_val, y_val)` or `(x_val, y_val, val_sample_weights)` with variable parameters that pass the validation data to the model, and optionally, the `val_sample_weights` as well. This also overrides `validation_split`, so use one or the other.
- **shuffle:** A Boolean that tells the model whether or not to shuffle the training data before each epoch, or pass in a string for “`batch`”, meaning it shuffles in batch-sized chunks.
- **class_weight:** (optional) A dictionary that tells the model how to weigh certain classes in the training process. You can use it to weigh under-represented classes higher, for example.
- **sample_weight:** (optional) A Numpy array of weights that have a 1:1 map between the training samples and the weight array you passed in. If you have temporal data (an extra time dimension), pass in a 2D

array with a shape (samples, sequence_length) to apply these weights to each timestep of the samples. Don't forget to set "temporal" for `sample_weight_mode` in `model.compile()`.

- **initial_epoch:** An integer that tells the model what epoch to start training at (can be used when resuming training).
- **steps_per_epoch:** The number of steps, or batches of samples, for the model to take before completing one epoch.
- **validation_steps:** (Only if you specify `steps_per_epoch`.) The number of steps to take (number of batches of samples) to use for validation before stopping.
- **validation_freq:** (Only if you pass in validation data.) If you pass in **n**, it runs validation every **n** epochs. If you pass in [**a**, **e**, **h**], it runs validation after epoch **a**, epoch **e**, and epoch **h**.

Model Evaluation and Prediction

After training the model, you can not only evaluate its performance on some test data, but you can make predictions and use the output for any other application you want. Previously, you've used the predictions to generate AUC scores to help better evaluate the model (accuracy is not the best metric to judge model performance by), but you can use these predictions in any way you want, especially if the model's really good at its job.

The code to evaluate your model on some test data might look similar to Figure A-8.

```
In [ ]: 1 model.evaluate(x, y, verbose=0)
```

Figure A-8. Code to evaluate the model given *x* and *y* data sets

For `model.evaluate()`, the parameters are

- **x:** The Numpy array representing the test data. Pass in a list of Numpy arrays if the model has multiple inputs.
- **y:** The Numpy array of target or label data that is a part of the test data. If there are multiple inputs, pass in a list of Numpy arrays.

- **batch_size:** If none is specified, the default is 32. This parameter expects an integer value that dictates how many samples there are per evaluation step.
- **verbose:** If set to 0, no output is shown. If set to 1, the progress bar is shown and looks like Figure A-9.

```
In [19]: 1
          2 score = TCN.evaluate(x_test, y_test, verbose=1)
          3 print('Test loss:', score[0])
          4 print('Test accuracy:', score[1])

4197/4197 [=====] - 1s
Test loss: 0.06095811567112381
Test accuracy: 0.9914224446032881
```

Figure A-9. The evaluate function with verbosity 1

- **sample_weight:** (optional) A Numpy array of weights for each of the test samples. Again, either a 1:1 map between the sample and the weights, unless it's temporal data. If you have temporal data (an extra time dimension), pass in a 2D array with a shape (samples, sequence_length) to apply these weights to each timestep of the samples. Don't forget to set "temporal" for sample_weight_mode in model.compile().
- **steps:** If None, then ignored. Otherwise, it's the integer parameter n number of steps (batches of samples) before declaring the evaluation as done.
- **callbacks:** Works the same way as the callbacks parameter for model.fit().

Finally, to make predictions, you can run code similar to Figure A-10.

```
In [ ]: 1 model.predict(x)
```

Figure A-10. The prediction function generates predictions given some data set x

In this case, the parameters are

- **x**: The Numpy array representing the prediction data. Pass in a list of Numpy arrays if the model has multiple inputs.
- **batch_size**: If none is specified, the default is 32. This parameter expects an integer value that dictates how many samples there are per batch.
- **verbose**: Either a 0 or 1.
- **steps**: How many steps to take (batches of samples) before finishing the prediction process. This is ignored if None is passed in.
- **callbacks**: Works the same way as the callbacks parameter for `model.fit()`.

One more thing to mention: If you've saved a model, you can load it again by calling the code in Figure A-11.

```
In [ ]: 1 from keras.models import load_model
        2
        3 model = load_model('filepath.h5')
```

Figure A-11. Loading a model given some file path

Now that we've covered the basics of model construction and operation, let's move on to the parts that constitute the models themselves: **layers**.

Layers

Input Layer

`keras.layers.Input()`

This is the input layer of the entire model, and it has several parameters:

- **shape**: This is the shape tuple of integers that tells the layer what shape to expect. For example, if you pass in `shape=(input_shape)` and `input_shape` is `(31, 1)`, you're telling the model to expect entries that each have a dimension `(31, 1)`.

- **batch_shape:** This is also a shape tuple of integers that includes the batch size. Passing in `batch_shape = (input_shape)`, where `input_shape` is `(100, 31, 1)`, tells the model to expect batches of 100 31x1 dimensional entries. Passing in an `input_shape` of `(None, 31, 1)` tells the model that the number of batches can be some arbitrary number.
- **name:** (Optional) A string name for the layer. It must be unique, and if nothing is passed in, some name is autogenerated.
- **dtype:** The data type that the layer should expect the input data to have, specified as a string. It can be something like `'int32'`, `'float32'`, etc.
- **sparse:** A Boolean that tells the layer whether or not the placeholder that the layer creates is sparse.
- **tensor:** (Optional) A tensor to pass into the layer to serve as the placeholder for input. If something is passed in, then Keras will not automatically create some placeholder tensor.

Dense Layer

`keras.layers.Dense()`

This is a neural network layer comprised of densely-connected neurons. Basically, every node in this layer is fully connected with the previous and next layers if there are any.

Here are the parameters:

- **units:** The number of neurons in this layer. This also factors into the dimension of the output space.
- **activation:** The activation function to use for this layer.
- **use_bias:** A Boolean for whether or not to use a bias vector in this layer.
- **kernel_initializer:** An initializer for the weight matrix. For more information, check out the **Initializers** section.
- **bias_initializer:** Similar to the `kernel_initializer`, but for the bias.

- **kernel_regularizer**: A regularizer function that's been applied to the weight matrix. For more information, check out the **Regularizers** section.
- **bias_regularizer**: Regularizer function applied to the bias.
- **activity_regularizer**: Regularizer function applied to the output of the layer.
- **kernel_constraint**: A constraint function applied to the weights. For more information, check out the **Constraints** section.
- **bias_constraint**: A constraint function applied to the bias.

For a better idea of what a dense layer is, check out [Figure A-12](#).

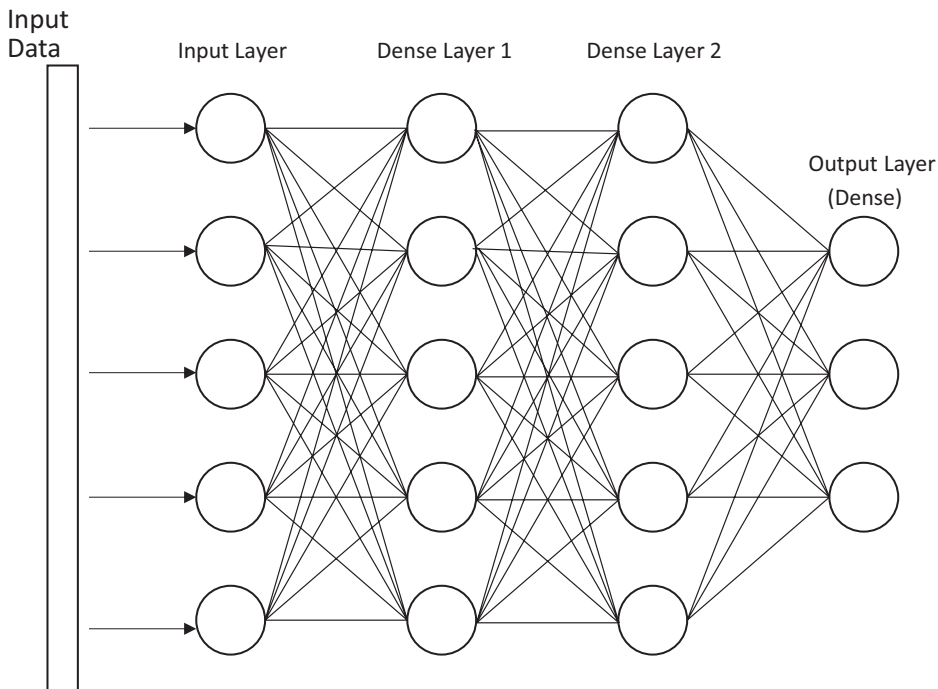


Figure A-12. Dense layers in an artificial neural network

Activation

`keras.layers.Activation()`

This layer applies an activation function to the input. Here is the argument:

- **activation:** Pass in either the activation function (see the **Activations** section) or some Theanos or TensorFlow operation.

To understand what an activation function is, Figure A-13 shows what each artificial neuron looks like.

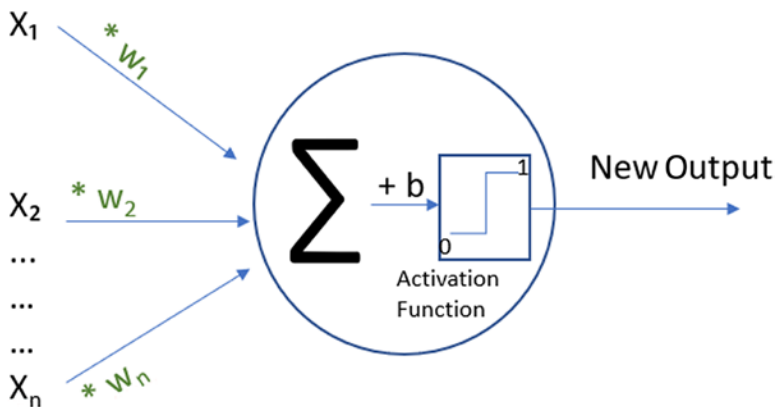


Figure A-13. The activation function is applied to the output of the function the node carries out on the input

The activation passes in the output from the input * weights + bias and passes it into the activation function. If there is no activation function, then that input just gets passed along as the output.

Dropout

`keras.layers.Dropout()`

What the dropout layer does is take some float **f** proportion of nodes in the preceding layer and “deactivates” them, meaning they don’t connect to the next layer. This can help combat overfitting on the training data.

Here are the parameters:

- **rate**: A float value between 0 and 1 that indicates the proportion of input units to drop.
- **noise_shape**: A 1D integer binary tensor that is multiplied with the input to determine what units are turned on or off. Instead of randomly selecting values using **rate**, you can pass in your own dropout mask to use in the dropout layer.
- **seed**: An integer to use as a random seed.

Flatten

```
keras.layers.Flatten()
```

This layer takes all of the inputs and flattens them into a single dimension.

Images can have three channels if they're color images. They can be RGB (red, green, blue), BGR (blue, green, red), HSV (hue, saturation, value), etc., so the dimensions of these images are actually (height, width, channels) if it's formatted channels last or (channels, height, width) if it's formatted channels first. To preserve this formatting, there is a parameter you can pass in to the flatten layer:

- **data_format**: A string that's either 'channels_first' or 'channels_last'. This tells the flattening layer how to format the flattened output to preserve this formatting.

To get a better idea of how the layer flattens the input, check out the summary in Figure [A-14](#) of a convolutional neural network.

```
In [5]: 1 import keras
2 from keras.models import Model
3 from keras.layers import Input, Dense, Convolution2D, Flatten
4
5 input_layer = Input(shape=(32, 32, 3))
6
7 conv_1 = Convolution2D(128, kernel_size=2, padding='same', activation='relu')(input_layer)
8
9 conv_2 = Convolution2D(128, kernel_size=2, padding='same', activation='relu')(conv_1)
10
11 flat_1 = Flatten()(conv_2)
12
13 classes = Dense(10, activation='softmax')(flat_1)
14
15 conv_net = Model(input_layer, classes)
16
17 conv_net.summary()
18
19
20
21
22
```

Layer (type)	Output Shape	Param #
input_5 (InputLayer)	(None, 32, 32, 3)	0
conv2d_6 (Conv2D)	(None, 32, 32, 128)	1664
conv2d_7 (Conv2D)	(None, 32, 32, 128)	65664
flatten_2 (Flatten)	(None, 131072)	0
dense_2 (Dense)	(None, 10)	1310730
Total params: 1,378,058		
Trainable params: 1,378,058		
Non-trainable params: 0		

Figure A-14. Notice how the flattening layer reduces the dimensionality of its input

Spatial Dropout 1D

`keras.layers.SpatialDropout1D()`

This function drops entire 1D feature maps instead of neuron elements, but otherwise has the same functionality as the regular dropout function. In earlier convolutional layers, the feature maps tend to be strongly correlated, so regular dropout functions won't help much with regularization in that case. Spatial dropout helps address this and also helps improve independence between the feature maps themselves.

The function takes one parameter:

- **rate:** A float between 0 and 1 that determines the proportion of input units to drop.

Spatial Dropout 2D

`keras.layers.SpatialDropout2D()`

This function is similar to the spatial dropout 1D function, except it works on 2D feature maps. Images can have three channels if they're color images. They can be RGB (red, green, blue), BGR (blue, green, red), HSV (hue, saturation, value), etc., so the dimensions of these images are actually (height, width, channels) if it's formatted channels last or (channels, height, width) if it's formatted channels first.

This function takes one additional parameter compared to `SpatialDropout1D()`:

- **rate:** A float between 0 and 1 that determines the proportion of input units to drop.
- **data_format:** 'channels_first' or 'channels_last'. This tells the flattening layer how to format the flattened output to preserve the formatting of channels first or channels last.

Conv1D

`keras.layers.Conv1D()`

Check out Chapter 7 for a detailed explanation on how one-dimensional convolutions work.

This layer is a one-dimensional (or temporal) convolutional layer. It basically passes a filter over the one-dimensional input and multiplies the values element-wise to create the output feature map.

These are the parameters that the function takes:

- **filters:** An integer value that determines the dimensionality of the output space. In other words, this is also the number of filters in the convolution.
- **kernel_size:** An integer (or tuple/list of a single integer) that specifies the length of the filter/kernel that is used in the 1D convolution.
- **strides:** An integer (or tuple/list of a single integer) that tells the layer how many data entries to shift by after one element-wise multiplication of the filter and the input data. Note: A stride value != 1 isn't compatible if the `dilation_rate != 1`.

- **padding:** ‘valid’, ‘causal’, or ‘same’. ‘valid’ doesn’t zero pad the output. ‘same’ zero pads the output so that it’s the same length as the input. ‘causal’ padding generates causal, dilated convolutions. For an explanation on what ‘causal’ padding is, refer to Chapter 7.
- **data_format:** ‘channels_first’ or ‘channels_last’. This tells the flattening layer how to format the flattened output to preserve the formatting of channels first or channels last. ‘channels_first’ has the format (batch, features, steps), and ‘channels_last’ has the format (batch, steps, features).
- **dilation_rate:** An integer (or tuple/list of a single integer) serves as the dilation rate for this dilated convolutional layer. For an explanation of how this works, refer to Chapter 7.
- **activation:** Passes in either the activation function (see the **Activations** section) or some Theano or TensorFlow operation. If nothing is specified, the data is passed along unaltered after the convolutional process.
- **use_bias:** A Boolean for whether or not to use a bias vector in this layer.
- **kernel_initializer:** An initializer for the weight matrix. For more information, check out the **Initializers** section.
- **bias_initializer:** Similar to the kernel_initializer, but for the bias.
- **kernel_regularizer:** A regularizer function that’s been applied to the weight matrix. For more information, check out the **Regularizers** section.
- **bias_regularizer:** A regularizer function applied to the bias.
- **activity_regularizer:** A regularizer function applied to the output of the layer.
- **kernel_constraint:** A constraint function applied to the weights. For more information, check out the **Constraints** section.
- **bias_constraint:** A constraint function applied to the bias.

Conv2D

`keras.layers.Conv1D()`

Check out Chapter 3 for a detailed explanation on how the 2D convolutional layer works.

This layer is a two-dimensional convolutional layer. It basically passes a 2D filter over the input and multiplies the values element-wise to create the output feature map.

These are the parameters that the function takes:

- **filters:** An integer value that determines the dimensionality of the output space. In other words, this is also the number of filters in the convolution.
- **kernel_size:** An integer (or tuple/list of two integers) that specifies the height and width of the filter/kernel that is used in the 2D convolution.
- **strides:** An integer (or tuple/list of two integers, one for height and one for width, respectively) that tells the layer how many data entries to shift by after one element-wise multiplication of the filter and the input data. Note: A stride value $\neq 1$ isn't compatible if the `dilation_rate` $\neq 1$.
- **padding:** 'valid' or 'same.' 'valid' doesn't zero pad the output. 'same' zero pads the output so that it's the same length as the input.
- **data_format:** 'channels_first' or 'channels_last'. This tells the flattening layer how to format the flattened output to preserve the formatting of channels first or channels last.
- **dilation_rate:** An integer (or tuple/list of a two integers) serves as the dilation rate for this dilated convolutional layer. For an explanation of how this works, refer to Chapter 7.
- **activation:** Passes in either the activation function (see the **Activations** section) or some Theanos or TensorFlow operation. If nothing is specified, the data is passed along unaltered after the convolutional process.
- **use_bias:** A Boolean for whether or not to use a bias vector in this layer.

- **kernel_initializer:** An initializer for the weight matrix. For more information, check out the **Initializers** section.
- **bias_initializer:** Similar to the `kernel_initializer`, but for the bias.
- **kernel_regularizer:** A regularizer function that's been applied to the weight matrix. For more information, check out the **Regularizers** section.
- **bias_regularizer:** A regularizer function applied to the bias.
- **activity_regularizer:** A regularizer function applied to the output of the layer.
- **kernel_constraint:** A constraint function applied to the weights. For more information, check out the **Constraints** section.
- **bias_constraint:** A constraint function applied to the bias.

UpSampling 1D

`keras.layers.UpSampling1D()`

For a detailed explanation on how upsampling works, refer to Chapter 7.

This layer essentially repeats the data **n** times with respect to time (where **n** is the parameter passed in):

- **size:** An integer **n** that specifies how many times to repeat each data entry with respect to time. The order of time is preserved, so each element is repeated **n** times according to its time entry.

UpSampling 2D

`keras.layers.UpSampling2D()`

Similar to `UpSampling1D()`, but for 2D inputs. The rows and columns are repeated **n** times according to `size[0]` and `size[1]`.

This is the list of parameters:

- **size:** An integer or tuple of two integers. The integer is the upsampling factor for both rows and columns, and the tuple lets you specify the upsampling factor for rows and for columns individually.

- **data_format:** ‘channels_first’ or ‘channels_last’. This tells the flattening layer how to format the flattened output to preserve the formatting of channels first or channels last.
- **interpolation:** ‘nearest’ or ‘bilinear’. CNTK does not support ‘bilinear’ yet, and Theanos only supports size=(2,2). ‘nearest’ and ‘bilinear’ are interpolation techniques used in image processing.

ZeroPadding1D

`keras.layers.ZeroPadding1D()`

Depending on the input, pads the input sequence with zeroes on both sides or either a zero on the left side or a zero on the right side of the input sequence.

This is the list of parameters:

- **padding:** An integer, a tuple of two integers, or a dictionary. The integer is a number that tells the layer how many zeroes to add on both the left and right side. An input of **1** adds a zero on both the left and right side. The tuple is formatted as (`left_pad`, `right_pad`), so you can pass in (0, 1) to tell it to add no zeroes on the left side and add one zero on the right side.

ZeroPadding2D

`keras.layers.ZeroPadding2D()`

Depending on the input, it pads the input sequence with a row and columns of zeroes at the top, left, right, and bottom of the image tensor.

This is the list of parameters:

- **padding:** An integer, a tuple of two integers, a tuple of two tuples with two integers each. The integer tells it to add **n** rows of zeroes on the top and bottom of the image tensor, and **n** columns of zeroes. The tuple of two integers is formatted as (`symmetric_height_pad`, `symmetric_width_pad`), so you can tell the layer to add **m** rows of zeroes and **n** columns of zeroes to each side, respectively, if you pass in a tuple (**m**, **n**). Finally, the tuple of two tuples is formatted as ((`top_pad`, `bottom_pad`), (`left_pad`, `right_pad`)), so you can customize even more how you want the layer to add rows or columns of zeroes.

- **data_format:** 'channels_first' or 'channels_last'. This tells the flattening layer how to format the flattened output to preserve the formatting of channels first or channels last.

MaxPooling1D

`keras.layers.MaxPooling1D()`

It applies max pooling on a 1D input. To get a better idea of how max pooling works, check out Chapter 3. Max pooling in 1D is similar to max pooling in 2D, except the sliding window only works in one dimension, going from left to right.

This is the list of parameters:

- **pool_size:** An integer value. If an integer **n** is given, then the window size of the pooling layer is **1xn**. These are also the factors to downscale by, so if an integer **n** is passed in, the dimensions for both height and width are downscaled by that factor.
- **strides:** An integer or None. By default, the stride is set to **pool_size**. If you pass in an integer, the pooling window moves by integer **n** amount after completing its pooling operation on a set of entries.
- **padding:** 'valid' or 'same'. 'valid' means there's no zero padding, and 'same' pads the output sequence with zeroes so that it matches the dimensions of the input sequence.
- **data_format:** 'channels_first' or 'channels_last'. This tells the flattening layer how to format the flattened output to preserve the formatting of channels first or channels last. 'channels_first' has the format (batch, features, steps), and 'channels_last' has the format (batch, steps, features).

MaxPooling2D

`keras.layers.MaxPooling2D()`

It applies max pooling on a 2D input. To get a better idea of how max pooling works, check out Chapter 3.

This is the list of parameters:

- **pool_size:** An integer that dictates the size of the pooling window. An integer of **n** makes the pooling window size **n**, meaning it sifts through **n** entries at a time and selects the maximum value to pass on to the output.
- **strides:** An integer or None. By default, the stride is set to **pool_size**. If you pass in an integer, the pooling window moves by integer **n** amount after completing its pooling operation on a set of entries. It is also a factor that determines how much to downscale the dimensions by, as a parameter **n** will reduce the dimensions by a factor **n**.
- **padding:** ‘valid’ or ‘same.’ ‘valid’ means there’s no zero padding, and ‘same’ pads the output sequence with zeroes so that it matches the dimensions of the input sequence.
- **data_format:** ‘channels_first’ or ‘channels_last.’ This tells the flattening layer how to format the flattened output to preserve the formatting of channels first or channels last.

Loss Functions

In the examples, `y_true` is the true label and `y_pred` is the predicted label.

Mean Squared Error

```
keras.losses.mean_squared_error(y_true, y_pred)
```

If you have questions on the notation for this equation, refer to Chapter 3. See the equation in Figure A-15.

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n (h_{\theta}(x^i) - y^i)^2$$

Figure A-15. The equation for mean squared error

Given input θ , the weights, the formula finds the average difference squared between the predicted value and the actual value. The parameter h_θ represents the model with the weight parameter θ passed in, so $h_\theta(x^i)$ gives the predicted value for x^i with model's weights θ . The parameter y^i represents the actual prediction for the data point at index i . Lastly, there are n entries in total.

This loss metric can be used in autoencoders to help evaluate the difference between the reconstructed output and the original. In the case of anomaly detection, this metric can be used to separate the anomalies from the normal data points, since anomalies have a higher reconstruction error.

Categorical Cross Entropy

`keras.losses.categorical_crossentropy(y_true, y_pred)`

See the equation in Figure A-16.

$$J(\theta) = -\frac{1}{n} \sum_{i=0}^n y_i * \log(h_\theta(x_i)) + (1 - y_i) * \log(1 - h_\theta(x_i))$$

Figure A-16. The equation for categorical cross entropy

In this case, n is the number of samples in the whole data set. The parameter h_θ represents the model with the weight parameter θ passed in, so $h_\theta(x_i)$ gives the predicted value for x_i with model's weights θ . Finally, y_i represents the true label for data point at index i . The data needs to be regularized to be between 0 and 1, so for categorical cross entropy, it must be piped through a softmax activation layer. The categorical cross entropy loss is also called **softmax loss**.

Equivalently, you can write the previous equation as shown in Figure A-17.

$$J(\theta) = -\frac{1}{n} \sum_{i=0}^n \sum_{j=0}^m y_{ij} * \log(h_\theta(x_{ij}))$$

Figure A-17. Another way to write the equation for categorical cross entropy

In this case, m is the number of classes.

The categorical cross entropy loss is a commonly used metric in classification tasks, especially in computer vision with convolutional neural networks. **Binary cross entropy** is a special case of categorical cross entropy where the number of classes m is two.

Sparse Categorical Cross Entropy

```
keras.losses.sparse_categorical_crossentropy(y_true, y_pred)
```

Sparse categorical cross entropy is basically the same as categorical cross entropy, but the distinction between them is in how their true labels are formatted. For categorical cross entropy, the labels are **one-hot encoded**. For an example of this, refer to Figure A-18, if you had your `y_train` formatted originally as the following, with six maximum classes.

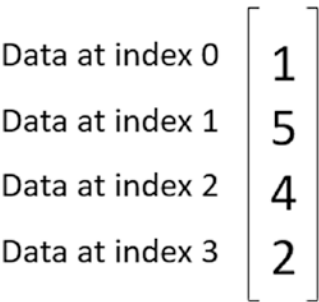


Figure A-18. An example of how `y_train` can be formatted. The value in each index is the class value that corresponds to the value at that index in `x_train`

You can call `keras.utils.to_categorical(y_train, n_classes)` with `n_classes` as 6 to convert `y_train` to that shown in Figure A-19.

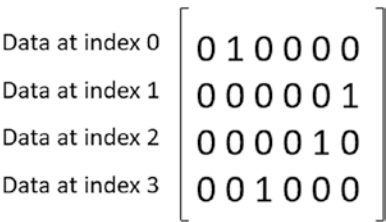


Figure A-19. The `y_train` in Figure A-18 is converted into a one-hot encoded format

So now your `y_train` looks like Figure A-20.

```

In [7]: 1 y_train = [1, 5, 4, 2]
        2
        3 keras.utils.to_categorical(y_train, 6)

Out[7]: array([[0., 1., 0., 0., 0., 0.],
               [0., 0., 0., 0., 0., 1.],
               [0., 0., 0., 0., 1., 0.],
               [0., 0., 1., 0., 0., 0.]])

```

Figure A-20. Converting `y_train` into a one-hot encoded format in Jupyter

This type of truth label formatting (**one-hot encoding**) is what categorical cross entropy uses. For sparse categorical cross entropy, it suffices to simply pass in the information in Figure A-21.

Data at index 0	1
Data at index 1	5
Data at index 2	4
Data at index 3	2

Figure A-21. The `y_train` to pass in for sparse categorical cross entropy

Or the code shown in Figure A-22.

```

In [7]: 1 y_train = [1, 5, 4, 2]
        2

```

Figure A-22. An example of `y_train` in the code that can be passed in if sparse categorical cross entropy is the metric

Metrics

Binary Accuracy

```
keras.metrics.binary_accuracy(y_true, y_pred)
```

To use this function, the ‘accuracy’ must be a metric that’s passed into the `model.compile()` function, and binary cross entropy must be the loss function.

Essentially, the function finds the number of instances where the true class label matches the rounded prediction label and finds the mean of the result (which is the same thing as dividing the total number of correct matches by the total number of samples).

The predicted values are rounded since as the neural network is trained more and more, the output values tend to change so that the predicted value is something really close to one, and the rest of the value are something really close to zero. In order to match the predicted values to the original truth labels (which are all integers), you can simply round the predicted values.

In the official Keras documentation on GitHub, this function is defined as shown in Figure A-23.

```

6  def binary_accuracy(y_true, y_pred):
7      '''Calculates the mean accuracy rate across all predictions for binary
8      classification problems.
9      ...
10     return K.mean(K.equal(y_true, K.round(y_pred)))

```

Figure A-23. The code definition in the Keras GitHub page of binary accuracy

Categorical Accuracy

`keras.metrics.categorical_accuracy(y_true, y_pred)`

Since most problems tend to involve categorical cross entropy (implying more than two classes in the data set), this tends to be the default accuracy metric when ‘accuracy’ is passed into the `model.compile()` function.

Instead of finding all of the instances where the true labels and rounded predictions match, categorical accuracy finds all of the instances where the true labels and predictions have a maximum value in the same spot.

Recall that for categorical cross entropy, the labels are one-hot encoded. Therefore, the truth labels only have one maximum per entry, along with the predictions (though again, one value will be really close to one while the others are really close to zero). What categorical accuracy does is check if the maximum value in the entry is in the same position for both `y_true` and for `y_pred`.

Once it’s found all those instances, it finds the mean of the result, leading to an accuracy value.

Essentially, it’s a similar equation to the one for binary accuracy, but with a different condition regarding `y_true` and `y_pred`.

The function is defined by Keras as shown in Figure A-24.

```

13 def categorical_accuracy(y_true, y_pred):
14     '''Calculates the mean accuracy rate across all predictions for
15     multiclass classification problems.
16     '''
17     return K.mean(K.equal(K.argmax(y_true, axis=-1),
18                           K.argmax(y_pred, axis=-1)))
19 
```

Figure A-24. The code definition of categorical accuracy as seen in the Keras GitHub page

Of course, there are many more metrics that are available on the Keras documentation, and you can even define **custom metrics**. To do that, just simply define a function that takes in `y_true` and `y_pred`, and call that function name in your metrics, as shown in Figure A-25.

```

In [ ]: 1 import keras.backend as K
2
3 def custom_metric(y_true, y_pred):
4     matches = K.equal(y_true, K.round(y_pred))
5     score = K.mean(matches)
6     return score
7
8 model.compile(optimizer='optimizer',
9               loss='loss_function',
10              metrics=['accuracy', custom_metric])

```

Figure A-25. Code to define a custom metric and use that for the model

In this example, you simply rewrite the binary accuracy metric in several lines and return the score. You can actually condense this function to just one line like in the actual implementation seen above, but this is just an example to showcase a custom metric.

Optimizers

SGD

`keras.optimizers.SGD()`

This is the **stochastic gradient descent** optimizer, a type of algorithm that aids in the backpropagation process by adjust the weights. It is commonly used as a training algorithm in a variety of machine learning applications, including neural networks.

The optimizer has several parameters:

- **lr:** Some float value where the learning rate $lr \geq 0$. The learning rate is a hyperparameter that determines how big of a step to take when optimizing the loss function.
- **momentum:** Some float value where the momentum $m \geq 0$. This parameter helps accelerate the optimization steps in the direction of the optimization, and helps reduce oscillations when the local minimum is overshoot (refer to Chapter 3 to refresh your understanding on how a loss function is optimized).
- **decay:** Some float value where the decay $d \geq 0$. Helps determine how much the learning rate decays by after each update (so that as the local minimum is approached, or after some number of training iterations, the learning rate decreases so smaller step sizes are taken. Big learning rates means the local minimum might be overshoot more easily).
- **nesterov:** A Boolean value to determine whether or not to apply Nesterov momentum. Nesterov momentum is a variation of momentum where the gradient is computed not from the current position, but from a position that takes into account the momentum. This is because the gradient always points in the right direction, but the momentum might carry the position too far forward and overshoot. Since it doesn't use the current position but instead some intermediate position that takes into account momentum, the gradient from that position can help correct the current course so that the momentum doesn't carry the new weights too far forward.

It essentially helps for more accurate weight updates and helps converge faster.

Adam

```
keras.optimizers.Adam()
```

The Adam optimizer is an algorithm that extends upon SGD, and has grown quite popular in deep learning applications in computer vision and in natural language processing.

These are the parameters for the algorithm:

- **lr**: Some float value where the learning rate $lr \geq 0$. The learning rate is a hyperparameter that determines how big of a step to take when optimizing the loss function. The paper describes good results with a value of 0.001 (the paper refers to the learning rate as **alpha**).
- **beta_1**: Some float value where $0 < \text{beta}_1 < 1$. This is usually some value close to 1, but the paper describes good results with a value of 0.9.
- **beta_2**: Some float value where $0 < \text{beta}_2 < 1$. This is usually some value close to 1, but the paper describes good results with a value of 0.999.
- **epsilon**: Some float value where epsilon $\epsilon \geq 0$. If None, then it defaults to `K.epsilon()`. Epsilon is some small number, described as $10E-8$ in the paper, to help prevent division by 0.
- **decay**: Some float value where the decay $d \geq 0$. Helps determine how much the learning rate decays by after each update (so that as the local minimum is approached, or after some number of training iterations, the learning rate decreases so smaller step sizes are taken. Big learning rates means the local minimum might be overshoot more easily).
- **amsgrad**: A Boolean on whether or not to apply the AMSGrad version of this algorithm. For more details on the implementation of this algorithm, check out “On the Convergence of Adam and Beyond.”

RMSprop

`keras.optimizers.RMSprop()`

RMSprop is a good algorithm for recurrent neural networks. RMSprop is a gradient-based optimization technique developed to help address the problem of gradients becoming too large or too small. RMSprop helps combat this problem by normalizing the gradient itself using the average of the squared gradients. In Chapter 7, it’s explained that one of the problems with RNNs is the vanishing/exploding gradient problem, leading to the development of LSTMs and GRU networks. And so it’s of no surprise that RMSprop pairs well with recurrent neural networks.

Besides the learning rate, it's recommended to leave the rest of the algorithms in their default settings. With that in mind, here are the parameters for this optimizer:

- **lr**: Some float value where the learning rate **lr** ≥ 0 . The learning rate is a hyperparameter that determines how big of a step to take when optimizing the loss function.
- **rho**: Some float value where **rho** ≥ 0 . Rho is a parameter that helps calculate the exponentially weighted average over the gradients squared.
- **epsilon**: Some float value where epsilon **e** ≥ 0 . If None, then it defaults to `K.epsilon()`. Epsilon is a very small number that helps prevent division by 0 and to help prevent the gradients from blowing up in RMSprop.
- **decay**: Some float value where the decay **d** ≥ 0 . Helps determine how much the learning rate decays by after each update (so that as the local minimum is approached, or after some number of training iterations, the learning rate decreases so smaller step sizes are taken. Big learning rates means the local minimum might be overshoot more easily).

Activations

You can pass in something like 'activation_function' for the activation parameter in a layer, or the full function, `keras.activations.activation_function()`, if you want to customize it more. Otherwise, the default initialized activation function is used in the layer.

Softmax

`keras.activations.softmax()`

This performs a softmax activation on the input **x** and on the given axis.

The two parameters are

- **x**: The input tensor
- **axis**: The axis that you want to use softmax normalization on. By default, it is set to -1.

The general formula for softmax is shown in Figure A-26 (K is the number of samples).

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^K e^{x_j}} \quad \text{For } i = 1, \dots, K \text{ and } x = (x_1, \dots, x_K) \in R^K$$

Figure A-26. The general formula for softmax

ReLU

`keras.activations.relu()`

ReLU, or “Rectified Linear Unit”, performs a simple activation based on the function shown in Figure A-27.

$$f(x) = \max(0, x)$$

Figure A-27. This is the general ReLU formula

The parameters are as follows:

- **x:** The input tensor
- **alpha:** A float that determines the slope of the negative part. Set to zero by default.
- **max_value:** A float value that represents the upper threshold, and is set to None by default.
- **threshold:** A float value set to 0.0 by default that’s the lower threshold.

If **max_value** is set, then you get the equation shown in Figure A-28.

$$f(x) = \max\_value \quad \text{for } x \geq \max\_value$$

Figure A-28. The ReLU formula if max_value is set

If **threshold** is also set, then you get the equation shown in Figure A-29.

$$f(x) = x \quad \text{for } \text{threshold} \leq x < \max\_value$$

Figure A-29. The ReLU formula if threshold is also set

Otherwise you get the equation shown in Figure A-30.

$$f(x) = \alpha * (x - \text{threshold})$$

Figure A-30. The formula for ReLU if α and threshold are set

For an example of what the base ReLU function does, refer to Figure A-31.

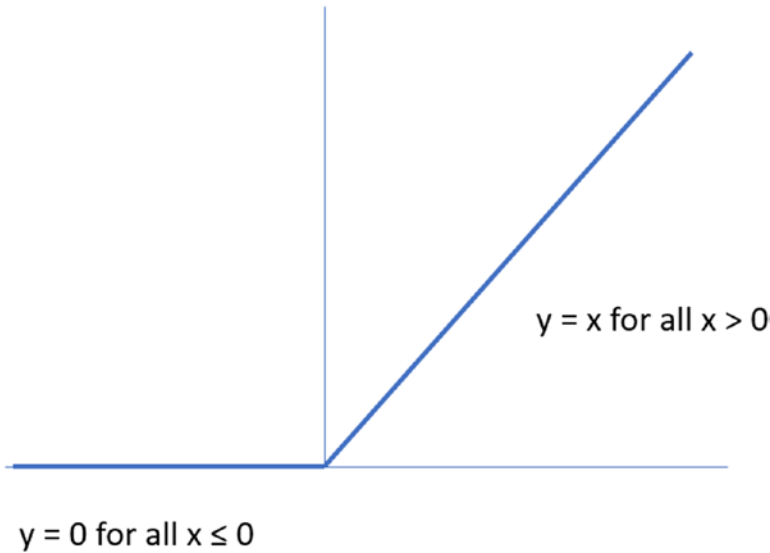


Figure A-31. The graph for a basic ReLU function

Sigmoid

```
keras.activations.sigmoid(x)
```

This is a simple activation function to call, as there are no parameters other than the input tensor $\mathbf{x}$.

The sigmoid function does have its uses, primarily because it forces the input to be between 0 and 1, but it is prone to the vanishing gradient problem, and so it is seldom used in hidden layers.

To get an idea of what the equation is like when graphed, refer to Figure A-32.

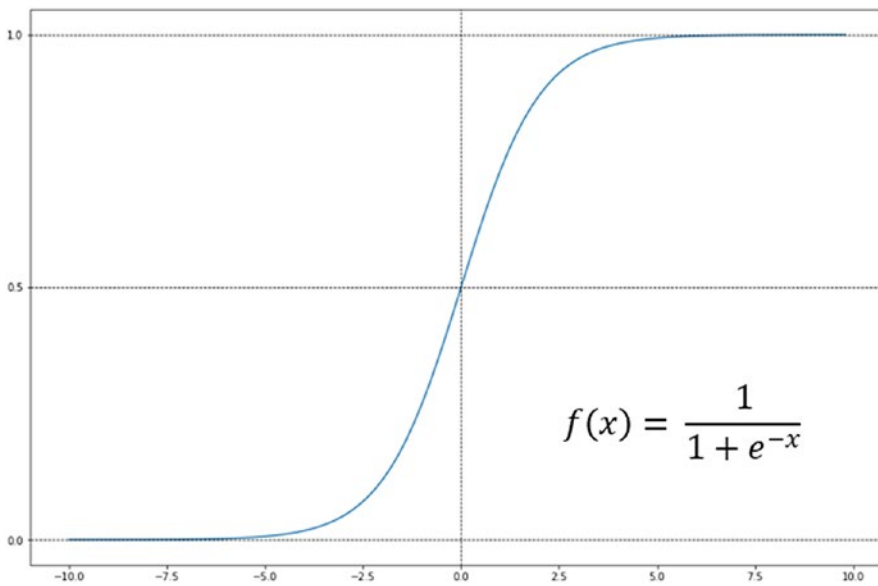


Figure A-32. The graph of a sigmoid function

Callbacks

ModelCheckpoint

`keras.callbacks.ModelCheckpoint()`

`ModelCheckpoint` is basically a function that saves the model every epoch (unless otherwise directed via parameters). How it does so can be configured by the set of parameters associated with `ModelCheckpoint()`:

- **filepath:** The path where you want to save the model file. Typing just “model_name.h5” saves it in the same directory.
- **monitor:** The quantity that you want the model to monitor. By default, it’s set to “val_loss”.
- **verbose:** Sets verbosity to 0 or 1. It’s set to 0 by default.
- **save_best_only:** If set to true, then the model with the best performance according to the quantity monitored will be saved.
- **save_weights_only:** If set to True, then only the weights will be saved. Essentially, if True, `model.save_weights(filepath)`; else, `model.save(filepath)`.

- **mode:** Choose between auto, min, or max. If `save_best_only` is True, then you should pick a choice that would suit the monitored quantity best. If you chose `val_acc` for monitor, then you want to pick max for mode, and if you choose `val_loss` for monitor, pick min for mode.
- **period:** How many epochs there are between each checkpoint.

TensorBoard

`keras.callbacks.TensorBoard()`

TensorBoard is a visualization tool that comes with TensorFlow. It helps you see in detail what's going on as your model trains.

To launch TensorBoard, type this into the command prompt:

```
tensorboard --logdir=/full_path_to_your_logs
```

```
keras.callbacks.TensorBoard(log_dir='./logs', histogram_freq=0, batch_
size=32, write_graph=True, write_grads=False, write_images=False,
embeddings_freq=0, embeddings_layer_names=None, embeddings_metadata=None,
embeddings_data=None, update_freq='epoch')
```

With that, here is the list of parameters:

- **log_dir:** The path to the directory where you want the model to save the log files. This is the same directory you pass as an argument in the command prompt. It is `./logs` by default.
- **histogram_freq:** The frequency (in epochs) that you want the activation and weight histograms to be computed for the model's layers. Set to 0 by default, which means it won't compute histograms. To visualize these histograms, `validation_data` (or `validation_split`) must be passed in.
- **batch_size:** The size of each batch of inputs to pass into the network to compute histograms from. Set to 32 by default.
- **write_graph:** Whether or not to allow the graph to be visualized in TensorBoard. Set to True by default. Note: When set to True, the log files can become large.

- **write_grads:** Whether or not to allow TensorBoard to visualize the gradient histograms. Set to False by default, and also needs `histogram_freq` to be a value greater than 0.
- **write_images:** Whether or not to visualize the model weights as an image in TensorBoard. Set to False by default.
- **embeddings_freq:** The frequency, in epochs, to save selected embedding layers. Set to 0 by default, which means that the embeddings won't be computed. To visualize data in TensorBoard's Embedding tab, pass in the data as `embeddings_data`.
- **embeddings_layer_names:** The list of names of layers for TensorBoard to track. If None or an empty list, then all of the layers will be watched. Set to None by default.
- **embeddings_metadata:** A dictionary that maps layer names to the corresponding file names where the metadata for this embedding layer is saved. Set to None by default. If the same metadata file is used for all of the embedding layers, then a string can be passed.
- **embeddings_data:** The data to be embedded at the layers specified in `embeddings_layer_names`. This is a Numpy array if the model expects a single input, and multiple Numpy arrays if the model has multiple inputs. Set to None by default.
- **update_freq:** A 'batch', 'epoch', or integer. 'batch' writes the losses and metrics to TensorBoard after each batch. 'epoch' is the same, except the losses and metrics are written to TensorBoard after each epoch. The integer tells it to write the metrics and losses to TensorBoard every integer `n` samples, where `n` is the integer passed in. Note: Writing to TensorBoard too frequently can slow down the training process.

With that being said, Figure A-33 shows an example of using TensorBoard as a callback when training a convolutional neural network on the MNIST data set.


```
tensorboard = keras.callbacks.TensorBoard(log_dir='./Graph',
histogram_freq=0,

        write_graph=True, write_images=True)

checkpoint =
keras.callbacks.ModelCheckpoint(filepath="keras_MNIST_CNN.h5",

        verbose=0,

        save_best_only=True)

model.fit(x_train, y_train,

        batch_size=batch_size,

        epochs=n_epochs,

        verbose=1,

        validation_data=(x_test, y_test),

        callbacks=[checkpoint, tensorboard])
```

Figure A-33. *Code to define a TensorBoard callback and use that when training*

Once you execute that code, you will notice the training process will begin. At this point, enter the line

```
tensorboard --logdir=/full_path_to_your_logs
```

into your command prompt and press Enter. It should show you something like [Figure A-34](#).

```
TensorBoard 1.10.0 at http://MSI:6006 (Press CTRL+C to quit)
```

Figure A-34. You should see something like this after executing the above line in command prompt. It should tell you where to go to access TensorBoard, which is `http://MSI:6006` in this case

Simply follow that link and you should see the screen shown in Figure A-35.

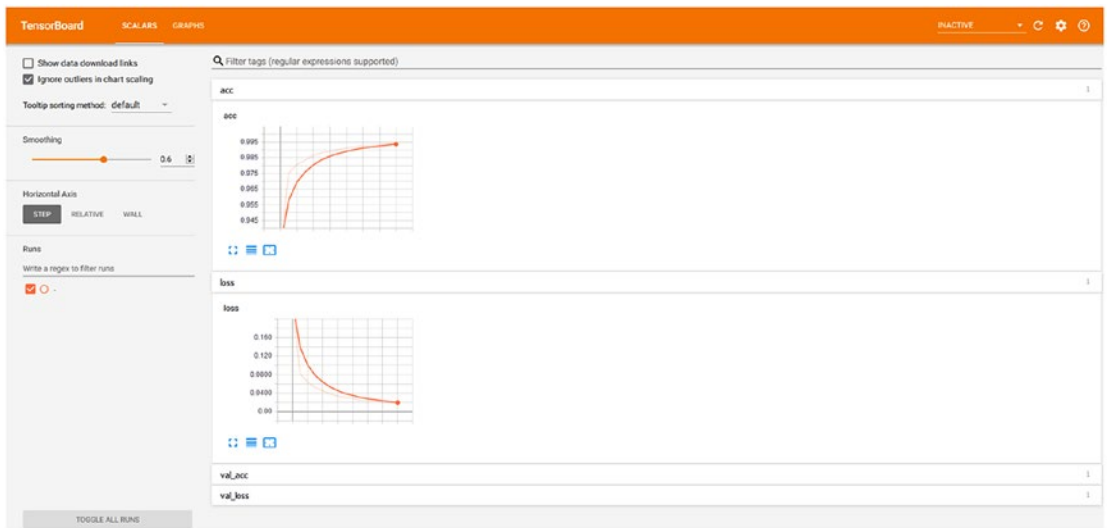


Figure A-35. The general page that appears when you launch TensorBoard

From here, you can see graphs for the metrics accuracy and loss. You can expand the other two metrics, `val_acc` and `val_loss`, to view those graphs as well (see Figure A-36).



Figure A-36. *Graphs for `val_acc` and `val_loss`*

As for the individual graphs, you can expand them out by pressing the leftmost button below the graph, and you can view data on the graph as you move your mouse across it, as seen in Figure A-37.

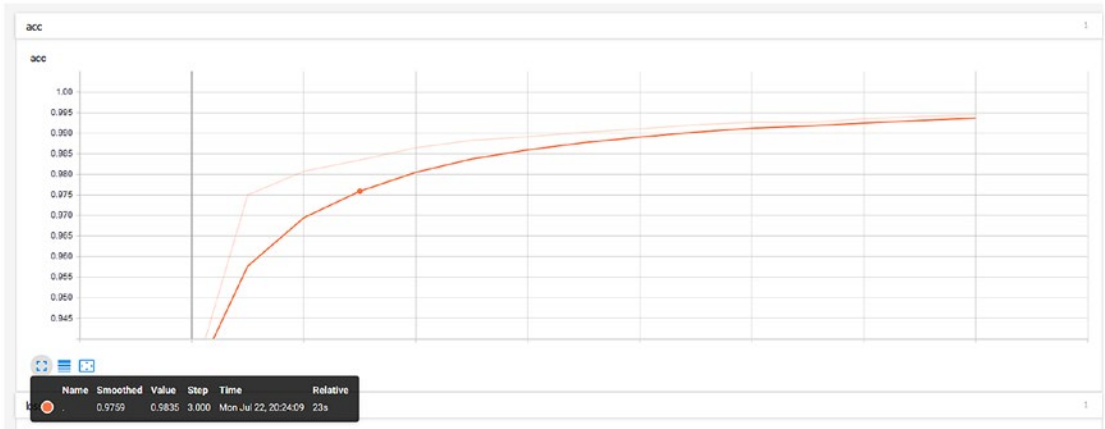


Figure A-37. The result of pressing the leftmost button underneath the graph. Doing so expands the graph, and regardless of whether the graph is expanded or not, you can point your mouse cursor at any point along the graph to get more details about that point

You can also view a graph of the entire model by pressing the Graphs tab, as shown in Figure A-38.



Figure A-38. There are two tabs. You started on the tab named SCALARS. Press GRAPHS to switch the tab

Doing so will result in a graph similar to the one shown in Figure A-39.

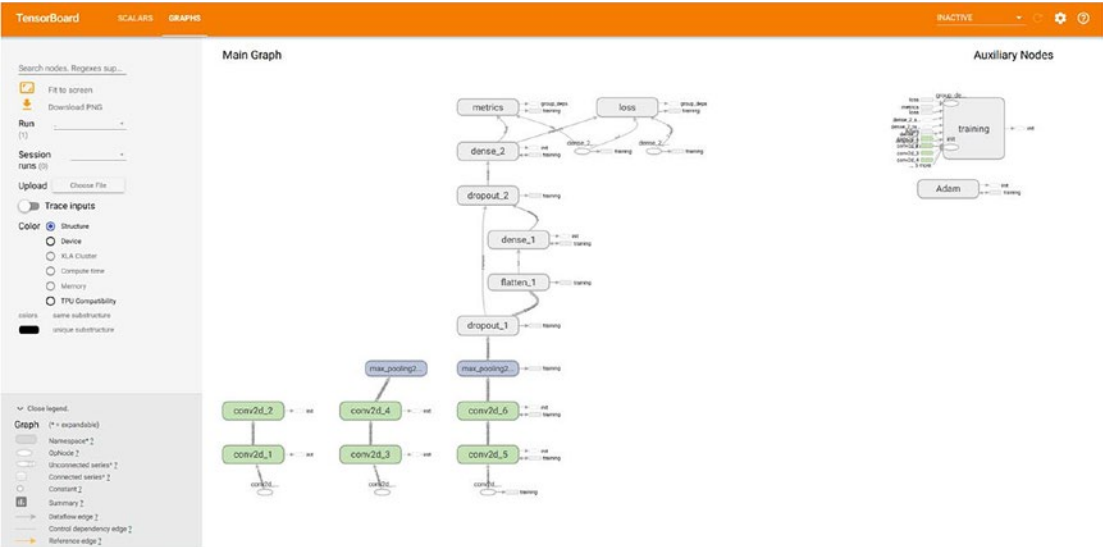


Figure A-39. The result of clicking on the GRAPHS tab

There are definitely more features and functionality that TensorBoard offers, but the general idea is that you will be able to examine your models in a much better fashion.

Back End (TensorFlow Operations)

You can also perform operations with TensorFlow (if it is the back end) through Keras by importing the back end. Below, we will demonstrate some basic functions, but keep in mind that TensorFlow has a vast variety of operations and functions.

You can use the back end to create custom layers, metrics, loss functions, etc., allowing for a much deeper level of customization. However, you must basically be knowledgeable in TensorFlow to accomplish all of this, since this is practically just using TensorFlow.

If you want the most customization possible, then using `tf.keras` along with TensorFlow is better, since `tf.keras` is wholly compatible with all of TensorFlow, and you'll have access to many more TensorFlow commands that you can't get with just the Keras back end.

Here are some of the commands you can execute using the back end (Figure A-40, Figure A-41, Figure A-42, Figure A-43).

```
In [125]: 1 import keras.backend as K
2
3 #Declaring a placeholder
4 a = K.placeholder(shape=(1,2,3)) # Equivalent to tf.placeholder()
5 print(a)
6
7 vals = [0, 1, 2, 3, 4, 5]
8 b = K.variable(value=vals) # Equivalent to tf.Variable()
9 print(b)

Tensor("Placeholder_62:0", shape=(1, 2, 3), dtype=float32)
<tf.Variable 'Variable_65:0' shape=(6,) dtype=float32_ref>
```

Figure A-40. Some TensorFlow operations such as defining placeholders and variables done through the Keras back end

```
In [126]: 1 import keras.backend as K
2
3 c = K.placeholder(shape=(1, 2))
4 d = K.placeholder(shape=(2, 5))
5
6 print(K.dot(c, d))

Tensor("MatMul_13:0", shape=(1, 5), dtype=float32)
```

Figure A-41. Finding the dot product of two placeholder variables *c* and *d* using the Keras back end

```
In [134]: 1 print(K.sum(c, axis=0))
2 print(K.sum(c, axis=1))

Tensor("Sum_7:0", shape=(2,), dtype=float32)
Tensor("Sum_8:0", shape=(1,), dtype=float32)
```

Figure A-42. Finding the sum of *c* along different axes using the Keras back end

```
In [136]: 1 print(K.mean(c, axis=0))

Tensor("Mean_1:0", shape=(2,), dtype=float32)
```

Figure A-43. Finding the mean of *c* using the Keras back end

Those are just some of the most basic functions available through the back end. The complete list of backend functions is available at <https://keras.io/backend/>.

Summary

Keras is a great tool to help you easily get involved with creating, training, and testing deep learning models, and provides a great deal of functionality while abstracting away the complicated syntax that TensorFlow has. Keras by itself can be sufficient, but as the content gets more advanced, it's better to have the level of customization and flexibility that TensorFlow or PyTorch offers. Keras allows you to use a wide variety of functions through the back end, allowing you to write custom layers, custom models, metrics, loss functions, and so on, but for the most customization and flexibility in how you want your neural networks to be (especially if you want to make completely new types of neural networks), then either `tf.keras` + TensorFlow or PyTorch would be better suited for your needs.

APPENDIX B

Intro to PyTorch

In this appendix, you will be introduced to the PyTorch framework along with the functionality that it offers. PyTorch is more involved than Keras is, and it is a lower-level framework (meaning there's more syntax, and elements aren't abstracted away from you like in Keras).

Regarding the setup, we use

- Torch version 0.4.1 (PyTorch)
- CUDA version 9.0.176
- cuDNN version 7.3.0.29

What Is PyTorch?

PyTorch is a deep learning library for Python, developed by artificial-intelligence researchers at Facebook and based on the Torch library. While PyTorch is also a low-level language like TensorFlow, it is easier to pick up because of the huge difference in syntax. TensorFlow has a much steeper learning curve, and you have to define a lot more elements than in PyTorch.

TensorFlow at the moment far surpasses PyTorch in how much community support it has, and this is primarily because PyTorch is a relatively new framework. Although you will find more resources for TensorFlow, more and more people are switching to PyTorch due to it being more intuitive while still offering practically the same functionality as TensorFlow (though TensorFlow does have some functions that PyTorch does not, you can easily implement those functions in PyTorch if you know what the logic is; an example of this is $\arctanh$ function).

In the end, it is mostly a matter of personal preference when deciding to use TensorFlow or PyTorch. Depending on the context of your work, one framework might be more suitable than the other.

That being said, PyTorch might be easier to use for research purposes, considering that it is easier to prototype in due to the lessened burden from the syntax. On the other hand, TensorFlow has more resources and the advantage of having TensorBoard. It is also better suited for cross-platform compatibility, since a model can be trained in Python but deployed in Java, for example, allowing for better scalability. If loading and saving models is a priority, perhaps TensorFlow is more suitable. Again, it all comes down to personal preference, since there's usually a workaround for many of the problems that both frameworks might face.

Using PyTorch

This section will be a bit different from the previous appendix. Here, we will demonstrate how some basic tensor operations are done, and then move on to illustrating how to use PyTorch by exploring PyTorch equivalent models of the temporal convolutional networks in Chapter 7.

First, let's begin by looking at some simple tensor operations. If you would like to know more about the framework itself and the functionality that it supports, check out the documentation at <https://pytorch.org/docs/0.4.1/index.html> and the code implementation at <https://github.com/pytorch/pytorch>.

Let's begin (see Figure B-1).

```

In [53]: 1 import torch
          2 import torch.nn
          3 import numpy as np
          4
          5 a = np.random.randint(0, 10, 5)
          6 a = torch.tensor(a)
          7 a

Out[53]: tensor([0, 3, 0, 9, 4], dtype=torch.int32)

In [54]: 1 b = torch.tensor(np.random.randint(0, 10, 5))
          2 b

Out[54]: tensor([3, 9, 7, 4, 2], dtype=torch.int32)

In [55]: 1 c = torch.add(a, b)
          2 c_summed = torch.sum(c)
          3
          4 c, c_summed

Out[55]: (tensor([ 3, 12,  7, 13,  6], dtype=torch.int32), tensor(41))

In [56]: 1 d = torch.tanh(b.float())
          2 d

Out[56]: tensor([0.9951, 1.0000, 1.0000, 0.9993, 0.9640])

In [57]: 1 torch.mean(d)

Out[57]: tensor(0.9917)

```

Figure B-1. A series of tensor operations in PyTorch. The code shows the operation and the output shows the results after the operations were performed on the corresponding tensors

With PyTorch, you can see that the data values like the tensors are some sort of array, unlike in TensorFlow. In TensorFlow, you must run the variable through a session to be able to see the data values.

In comparison, Figure B-2 shows TensorFlow.

```

In [78]: 1 import tensorflow as tf
          2 import numpy as np
          3
          4 f = tf.constant(np.random.randint(0, 10, 5), shape=(1, 5), dtype=tf.int32)
          5 print(f)
          6 g = tf.constant(np.random.randint(0, 10, 5), shape=(1, 5), dtype=tf.int32)
          7 print(g)
          8 result = A + B
          9 tanh = tf.tanh(tf.to_float(result))
         10
         11
         12 with tf.Session() as sess:
         13     print("f: {}    g: {}".format(sess.run(f), sess.run(g)))
         14     print("f + g: {}".format(sess.run(result)))
         15     print("tanh(f+g): {}".format(sess.run(tanh)))

Tensor("Const_37:0", shape=(1, 5), dtype=int32)
Tensor("Const_38:0", shape=(1, 5), dtype=int32)
f: [[5 6 9 7 0]]    g: [[3 9 4 8 9]]

f + g: [[14 16 6 5 16]]

tanh(f+g): [[1.          1.          0.99998784 0.99990916 1.          ]]

```

Figure B-2. Some tensor operations conducted in TensorFlow. Note that to actually see results, you need to pass everything through a TensorFlow session

PyTorch has much more functionality in how you can manipulate tensors, so it's worth checking out the documentation if you haven't.

Now, let's move on to creating a PyTorch model in a somewhat advanced, but organized format. Splitting up the definition of the model, the training process, and the testing process into their respective parts will help you understand how these models are created, trained, and evaluated.

You start by applying a convolutional neural network to the MNIST data set in order to showcase the more customizable format of training.

As usual, you begin with your imports (see Figure B-3 and Figure B-4).

```

import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
import torch.nn.functional as F
import numpy as np

device = torch.device('cuda:0' if torch.cuda.is_available()
else 'cpu')

```

Figure B-3. *Importing the basic modules needed to create your network*

```

In [1]: 1 import torch
        2 import torch.nn as nn
        3 import torchvision
        4 import torchvision.transforms as transforms
        5 import torch.optim as optim
        6 import torch.nn.functional as F
        7 import numpy as np
        8
        9 device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

```

Figure B-4. *The code in Figure B-3 in a Jupyter cell*

In Chapter 3, the code was introduced in a manner similar to basic Keras formatting, so you defined the hyperparameters and loaded your data sets (data loaders in this case) right after importing the modules you need.

Instead, you will now define the model (see Figure B-5 and Figure B-6).

```
class CNN(nn.Module):  
    def __init__(self):  
        super(CNN, self).__init__()  
        self.conv1 = nn.Conv2d(1, 32, 3, 1)  
        self.conv2 = nn.Conv2d(32, 64, 3, 1)  
        self.dense1 = nn.Linear(12*12*64, 128)  
        self.dense2 = nn.Linear(128, num_classes)  
  
    def forward(self, x):  
        x = F.relu(self.conv1(x))  
        x = F.relu(self.conv2(x))  
        x = F.max_pool2d(x, 2, 2)  
        x = F.dropout(x, 0.25)  
        x = x.view(-1, 12*12*64)  
        x = F.relu(self.dense1(x))  
        x = F.dropout(x, 0.5)  
        x = self.dense2(x)  
        return F.log_softmax(x, dim=1)
```

Figure B-5. *Defining the model*

```

In [2]: 1
        2 class CNN(nn.Module):
        3     def __init__(self):
        4         super(CNN, self).__init__()
        5         self.conv1 = nn.Conv2d(1, 32, 3, 1)
        6         self.conv2 = nn.Conv2d(32, 64, 3, 1)
        7         self.dense1 = nn.Linear(12*12*64, 128)
        8         self.dense2 = nn.Linear(128, num_classes)
        9
        10    def forward(self, x):
        11        x = F.relu(self.conv1(x))
        12        x = F.relu(self.conv2(x))
        13        x = F.max_pool2d(x, 2, 2)
        14        x = F.dropout(x, 0.25)
        15        x = x.view(-1, 12*12*64)
        16        x = F.relu(self.dense1(x))
        17        x = F.dropout(x, 0.5)
        18        x = self.dense2(x)
        19        return F.log_softmax(x, dim=1)
        20

```

Figure B-6. The code in Figure B-5 in a Jupyter cell

With that out of the way, you can define both the training and testing functions (see Figure B-7 and Figure B-8 for the training function, and Figure B-9 and Figure B-10 for the testing function).

```

def train(model, device, train_loader, criterion, optimizer, epoch,
          save_dir='model.ckpt'):
    total_step = len(train_loader)

    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss:
{: .4f}'.format(epoch+1, num_epochs, i+1, total_step, loss.item()))

    torch.save(model.state_dict(), 'pytorch_mnist_cnn.ckpt')

```

Figure B-7. The training algorithm. The for loop takes each pair of image and labels and passes them into the GPU as a tensor. They then go into the model, and the gradients are calculated. The information about the epoch and loss are then output

```

In [6]: 1 def train(model, device, train_loader, criterion, optimizer, epoch, save_dir='model.ckpt'):
        2     total_step = len(train_loader)
        3     for i, (images, labels) in enumerate(train_loader):
        4         images = images.to(device)
        5         labels = labels.to(device)
        6
        7         # Forward pass
        8         outputs = model(images)
        9         loss = criterion(outputs, labels)
        10
        11         # Backward and optimize
        12         optimizer.zero_grad()
        13         loss.backward()
        14         optimizer.step()
        15
        16         if (i+1) % 100 == 0:
        17             print ('Epoch [{} / {}], Step [{} / {}], Loss: {:.4f}'
        18                   .format(epoch+1, num_epochs, i+1, total_step, loss.item()))
        19
        20     torch.save(model.state_dict(), 'pytorch_mnist_cnn.ckpt')

```

Figure B-8. The code in Figure B-7 in a Jupyter cell

The training function takes in the following parameters:

- **model:** An instance of a model class. In this case, it's an instance of the CNN class defined above.
- **device:** This basically tells PyTorch what device (if the GPU is an option, which GPU to run on, and if not, the CPU is the device) to run on. In this case, you define the device right after the imports.
- **train_loader:** The loader for the training data set. In this case, you use a data_loader because that's how the MNIST data is formatted when importing from torchvision. This data loader contains the training samples for the MNIST data set.
- **criterion:** The loss function to use. Define this before calling the train function.
- **optimizer:** The optimization function to use. Define this before calling the train function.
- **epoch:** What epoch is running. In this case, you call the training function in a for loop while passing in the iteration as the epoch.

The testing function is shown in Figure B-9 and Figure B-10.


```

from sklearn.metrics import roc_auc_score

def test(model, device, test_loader):

    preds = []
    y_true = []
    # Test the model
    model.eval() # Set model to evaluation mode.
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in test_loader:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            detached_pred = predicted.detach().cpu().numpy()
            detached_label = labels.detach().cpu().numpy()
            for f in range(0, len(detached_pred)):
                preds.append(detached_pred[f])
                y_true.append(detached_label[f])

        print('Test Accuracy of the model on the 10000 test images:
        {:.2%}'.format(correct / total))

    preds = np.eye(num_classes)[preds]
    y_true = np.eye(num_classes)[y_true]
    auc = roc_auc_score(preds, y_true)
    print("AUC: {:.2%}".format (auc))

```

Figure B-9. The code for the testing algorithm. Once again, the for loop takes the image and label pairs and passes them through the model to get a prediction. Then, once every pair has a prediction, the AUC score is calculated

```

In [7]: 1 from sklearn.metrics import roc_auc_score
        2
        3 def test(model, device, test_loader):
        4
        5     preds = []
        6     y_true = []
        7     # Test the model
        8     model.eval() # Set model to evaluation mode.
        9     with torch.no_grad():
        10         correct = 0
        11         total = 0
        12         for images, labels in test_loader:
        13             images = images.to(device)
        14             labels = labels.to(device)
        15             outputs = model(images)
        16             _, predicted = torch.max(outputs.data, 1)
        17             total += labels.size(0)
        18             correct += (predicted == labels).sum().item()
        19             detached_pred = predicted.detach().cpu().numpy()
        20             detached_label = labels.detach().cpu().numpy()
        21             for f in range(0, len(detached_pred)):
        22                 preds.append(detached_pred[f])
        23                 y_true.append(detached_label[f])
        24
        25         print('Test Accuracy of the model on the 10000 test images: {:.2%}'.format(correct / total))
        26
        27         preds = np.eye(num_classes)[preds]
        28         y_true = np.eye(num_classes)[y_true]
        29         auc = roc_auc_score(preds, y_true)
        30         print("AUC: {:.2%}".format(auc))
        31

```

Figure B-10. The code in Figure B-9 in a Jupyter cell

Notice that you use the AUC score as part of the testing metric. You don't have to do this, but it might be a better indicator of the model's performance than plain accuracy, so it was included in this example.

The parameters the model takes in are

- **model:** An instance of a model class. In this case, it's an instance of the CNN class defined above.
- **device:** This basically tells PyTorch what device (if the GPU is an option, which GPU to run on, and if not, the CPU is the device) to run on. In this case, you define the device right after the imports.
- **test_loader:** The loader for the testing data set. In this case, you use a `data_loader` because that's how the MNIST data is formatted when importing from `torchvision`. This data loader contains the testing samples for the MNIST data set.

Now you can get to defining your hyperparameters and data loaders, and calling your train and test functions (Figures B-11 through B-13).

```
# Hyperparameters
num_epochs = 15
num_classes = 10
batch_size = 128
learning_rate = 0.001

# Load MNIST data set
train_dataset = torchvision.datasets.MNIST(root='.././data/',
                                           train=True,

                                           transform=transforms.ToTensor(),

                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='.././data/',
                                          train=False,

                                          transform=transforms.ToTensor())

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
```

Figure B-11. *Defining the hyperparameters, loading the MNIST data, and defining the training and testing set data loaders*

```
model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

## Training phase

for epoch in range(0, num_epochs):
    train(model, device, train_loader, criterion, optimizer, epoch)

## Testing phase

test(model, device, test_loader)
```

Figure B-12. *Initializing the model and passing it to the GPU, defining your criterion function (cross entropy loss), and defining your optimizer (the Adam optimizer). Then, the training and testing functions are called*

```

In [8]: 1 #Hyperparameters
2 num_epochs = 15
3 num_classes = 10
4 batch_size = 128
5 learning_rate = 0.001
6
7 #Load MNIST data set
8 train_dataset = torchvision.datasets.MNIST(root='../data/',
9                                             train=True,
10                                            transform=transforms.ToTensor(),
11                                            download=True)
12
13 test_dataset = torchvision.datasets.MNIST(root='../data/',
14                                           train=False,
15                                           transform=transforms.ToTensor())
16
17 #Data loader
18 train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
19                                           batch_size=batch_size,
20                                           shuffle=True)
21
22 test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
23                                           batch_size=batch_size,
24                                           shuffle=False)
25
26
27
28 model = CNN().to(device)
29 criterion = nn.CrossEntropyLoss()
30 optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
31
32
33 ## Training phase
34
35 for epoch in range(0, num_epochs):
36     train(model, device, train_loader, criterion, optimizer, epoch)
37
38 ## Testing phase
39
40
41 test(model, device, test_loader)

```

Figure B-13. What the code from Figures B-11 and B-12 should look like after pasting them into a Jupyter cell

After the training process, you get Figure B-14 and Figure B-15.

```
Epoch [1/15], Step [100/469], Loss: 0.1852
Epoch [1/15], Step [200/469], Loss: 0.0556
Epoch [1/15], Step [300/469], Loss: 0.1524
Epoch [1/15], Step [400/469], Loss: 0.0332
Epoch [2/15], Step [100/469], Loss: 0.0494
Epoch [2/15], Step [200/469], Loss: 0.1003
Epoch [2/15], Step [300/469], Loss: 0.0534
Epoch [2/15], Step [400/469], Loss: 0.0329
Epoch [3/15], Step [100/469], Loss: 0.0387
Epoch [3/15], Step [200/469], Loss: 0.0379
Epoch [3/15], Step [300/469], Loss: 0.0055
Epoch [3/15], Step [400/469], Loss: 0.0183
Epoch [4/15], Step [100/469], Loss: 0.0283
Epoch [4/15], Step [200/469], Loss: 0.0250
Epoch [4/15], Step [300/469], Loss: 0.0210
Epoch [4/15], Step [400/469], Loss: 0.0637
Epoch [5/15], Step [100/469], Loss: 0.0109
Epoch [5/15], Step [200/469], Loss: 0.0091
Epoch [5/15], Step [300/469], Loss: 0.0243
Epoch [5/15], Step [400/469], Loss: 0.0095
Epoch [6/15], Step [100/469], Loss: 0.0310
Epoch [6/15], Step [200/469], Loss: 0.0254
Epoch [6/15], Step [300/469], Loss: 0.0078
```

Figure B-14. The initial output of the training process

```
Epoch [12/15], Step [200/469], Loss: 0.0017
Epoch [12/15], Step [300/469], Loss: 0.0006
Epoch [12/15], Step [400/469], Loss: 0.0026
Epoch [13/15], Step [100/469], Loss: 0.0010
Epoch [13/15], Step [200/469], Loss: 0.0002
Epoch [13/15], Step [300/469], Loss: 0.0025
Epoch [13/15], Step [400/469], Loss: 0.0014
Epoch [14/15], Step [100/469], Loss: 0.0000
Epoch [14/15], Step [200/469], Loss: 0.0000
Epoch [14/15], Step [300/469], Loss: 0.0001
Epoch [14/15], Step [400/469], Loss: 0.0064
Epoch [15/15], Step [100/469], Loss: 0.0024
Epoch [15/15], Step [200/469], Loss: 0.0000
Epoch [15/15], Step [300/469], Loss: 0.0017
Epoch [15/15], Step [400/469], Loss: 0.0003
Test Accuracy of the model on the 10000 test images: 98.93%
AUC: 99.40%
```

Figure B-15. The training process has finished

Although in your Keras examples you didn't spread apart your training and testing functions (since they're just one line each), more complicated implementations of models involving custom layers, models, and so on can be formatted in a similar fashion to the PyTorch example above.

Hopefully, you understand a bit more on how to implement, train, and test neural networks in PyTorch.

Next, we will explain some of the basic functionality that PyTorch offers in terms of model layers (activations included), loss functions, and optimizers, and then you'll explore PyTorch applications of temporal convolutional neural networks to the data set found in Chapter 7.

Sequential vs. ModuleList

Similar to Keras, PyTorch has a couple different ways to define the model.

Sequentially, as in Figure B-16

```
In [ ]: 1 ## sequential
        2 import torch.nn as nn
        3
        4 model = nn.Sequential(
        5     nn.Conv2d(1, 32, 3, 1),
        6     nn.ReLU(),
        7     nn.Conv2d(32, 64, 3, 1),
        8     nn.Sigmoid()
        9 )
```

Figure B-16. A sequential model in PyTorch

This is similar to the sequential model in Keras, where you add layers one at a time and in order.

ModuleList, as in Figure B-17

```

10
11  ## ModuleList
12
13  class ModuleListModel(nn.Module):
14      def __init__(self):
15          super(ModuleListModel, self).__init__()
16          self.conv_1 = nn.Conv2d(1, 32, 3, 1)
17          self.conv_2 = nn.Conv2d(32, 64, 3, 1)
18          self.dense_1 = nn.Linear(64*64, 128)
19          self.output = nn.Linear(128, n_classes)
20
21
22      def forward(self, x):
23          x = nn.functional.relu(self.conv_1(x))
24          x = nn.functional.relu(self.conv_2(x))
25          x = nn.functional.max_pool2d(x, 2, 2)
26          x = nn.functional.dropout(x, 0.25)
27          x = x.view(-1, 64*64)
28          x = nn.functional.relu(self.dense_1(x))
29          x = nn.functional.dropout(x, 0.5)
30          x = nn.functional.log_softmax(self.output(x), dim=1)
31          return x
32
33
34  model = ModuleListModel().to(device)

```

Figure B-17. A model in PyTorch defined in a ModuleList format

This is similar to the functional model that you can build in Keras. This is a more customizable way to build your model, and allows you much more flexibility in how you want to build it too.

Layers

We've covered how to build the models, so let's look at examples of some common layers you can build.

Conv1d

`torch.nn.Conv1d()`

Check out Chapter 7 for a detailed explanation on how one-dimensional convolutions work.

This layer is a one-dimensional (or temporal) convolutional layer. It basically passes a filter over the one-dimensional input and multiplies the values element-wise to create the output feature map.

These are the parameters that the function takes:

- **in_channels:** The dimensionality of the input space; the number of input nodes.
- **out_channels:** The dimensionality of the output space; the number of output nodes.
- **kernel_size:** The dimensionality of the kernel/filter. An integer **n** makes the dimensions of the kernel **nxn**, and a tuple of two integers allows you to specify the exact dimensions (**height**, **width**).
- **stride:** The number of elements to shift right by after one filter/kernel operation. An integer **n** makes the kernel shift right by that amount. A tuple of two integers allows you to specify (**vertical_shift**, **horizontal_shift**). Default = 1.
- **padding:** The amount of zero padding to add to the layer in the output. An integer **n** pads **n** entries to the rows and columns. A tuple of two integers allows you to specify (**vertical_padding**, **horizontal_padding**). Default = 0.
- **dilation:** For an explanation on how dilation works, refer to Chapter 7. An integer **n** means a dilation factor of **n**. Default = 1.
- **groups:** Controls the connections between the input and output nodes. Groups=1 means all inputs correlate with all outputs. Groups=2 means there's really two convolutional layers side by side, so half the inputs go to half the outputs. Default = 1.
- **bias:** Whether or not to use bias. Default = True.

Conv2d

`torch.nn.Conv2d()`

Check out Chapter 3 for a detailed explanation on how the 2D convolutional layer works.

This layer is a two-dimensional convolutional layer. It basically passes a 2D filter over the input and multiplies the values element-wise to create the output feature map.

These are the parameters that the function takes:

- **in_channels:** The dimensionality of the input space; the number of input nodes.
- **out_channels:** The dimensionality of the output space; the number of output nodes.
- **kernel_size:** The dimensionality of the kernel/filter. An integer **n** makes the dimensions of the kernel **nxn**, and a tuple of two integers allows you to specify the exact dimensions (**height, width**).
- **stride:** The number of elements to shift right by after one filter/kernel operation. An integer **n** makes the kernel shift right by that amount. A tuple of two integers allows you to specify (**vertical_shift, horizontal_shift**). Default = 1.
- **padding:** The amount of zero padding to add to the layer in the output. An integer **n** pads **n** entries to the rows and columns. A tuple of two integers allows you to specify (**vertical_padding, horizontal_padding**). Default = 0.
- **dilation:** For an explanation on how dilation works, refer to Chapter 7. An integer **n** means a dilation factor of **n**. A tuple of two integers allows you to specify (**vertical_dilation, horizontal_dilation**). Default = 1.
- **groups:** Controls the connections between the input and output nodes. Groups=1 means all inputs correlate with all outputs. Groups=2 means there's really two convolutional layers side by side, so half the inputs go to half the outputs. Default = 1.
- **bias:** Whether or not to use bias. Default = True.

Linear

```
torch.nn.Linear()
```

This is a neural network layer comprised of densely-connected neurons. Basically, every node in this layer is fully connected with the previous and next layers if there are any.

Here are the parameters:

- **in_features**: The size of each input sample; number of inputs.
- **out_features**: The size of each output sample; number of outputs.
- **bias**: Whether or not to use bias. Default = True.

MaxPooling1D

```
torch.nn.MaxPool1d()
```

This applies max pooling on a 1D input. To get a better idea of how max pooling works, check out Chapter 3. Max pooling in 1D is similar to max pooling in 2D, except the sliding window only works in one dimension, going from left to right.

This is the list of parameters:

- **kernel_size**: The size of the pooling window. If an integer **n** is given, then the window size of the pooling layer is **1xn**.
- **stride**: Defaults to kernel_size if nothing is passed in. If you pass in an integer, the pooling window moves by integer **n** amount after completing its pooling operation on a set of entries.
- **padding**: An integer **n** representing the zero padding to add on both sides. Default = 0.
- **dilation**: Similar to the dilation factor in the convolutional layer, except with max pooling. Default = 1.
- **return_indices**: If set to True, it will return the indices of the max values along with the outputs. Default = False.
- **ceil_mode**: If set to True, it will use ceil instead of floor to compute the output shape. This comes into play because of the dimensionality reduction involved (a kernel size of **n** will reduce dimensionality by a factor of **n**).

MaxPooling2D

`torch.nn.MaxPool2d()`

It applies max pooling on a 2D input. To get a better idea of how max pooling works, check out Chapter 3.

This is the list of parameters:

- **kernel_size**: The size of the pooling window. If an integer **n** is given, then the window size of the pooling layer is **1xn**. A tuple of two integers allows you to specify the dimensions as (**height**, **width**).
- **stride**: Defaults to `kernel_size` if nothing is passed in. If you pass in an integer, the pooling window moves by integer **n** amount after completing its pooling operation on a set of entries. A tuple of two integers allows you to specify (**vertical_shift**, **horizontal_shift**).
- **padding**: An integer **n** representing the zero padding to add on both sides. A tuple of two integers allows you to specify (**vertical_padding**, **horizontal_padding**). Default = 0.
- **dilation**: Similar to the dilation factor in the convolutional layer, except with max pooling. An integer **n** means a dilation factor of **n**. A tuple of two integers allows you to specify (**vertical_dilation**, **horizontal_dilation**). Default = 1.
- **return_indices**: If set to True, it will return the indices of the max values along with the outputs. Default = False.
- **ceil_mode**: If set to True, it will use ceil instead of floor to compute the output shape. This comes into play because of the dimensionality reduction involved (a kernel size of **n** will reduce dimensionality by a factor of **n**).

ZeroPadding2D

`torch.nn.ZeroPad2d()`

Depending on the input, it pads the input sequence with a row and columns of zeroes at the top, left, right, and bottom of the image tensor.

Here is the parameter:

- **padding:** An integer or a tuple of four integers. The integer tells it to add **n** rows of zeroes on the top and bottom of the image tensor, and **n** columns of zeroes. The tuple of four integers is formatted as (padding_left, padding_right, padding_top, padding_bottom), so you can customize even more how you want the layer to add rows or columns of zeroes.

Dropout

`torch.nn.Dropout()`

What the dropout layer does in PyTorch is take the input and randomly zeroes the elements according to some probability **p** using samples from a Bernoulli distribution. This process is random, so with every forward pass through the model, different elements will be chosen to be zeroed. This process helps with regularization of layer outputs and helps combat overfitting.

Here are the parameters:

- **p:** The probability of an element to be zeroed. Default = 0.5
- **inplace:** If set to True, it will perform the operation in place. Default = False.

You can define this as a layer within the model itself, or apply dropout in the forward function like so:

`torch.nn.functional.Dropout(input, p = 0.5, training=False, inplace=False)`

Input is the previous layer, and **training** is a parameter that determines whether or not you want this dropout layer to function outside of training (such as during evaluation).

Figure B-18 shows an example of how you can use this layer in the forward function.

```
In [ ]: 1 def forward(self, x):
        2     x = nn.functional.relu(self.conv_1(x))
        3     x = nn.functional.dropout(x, 0.25)
        4     x = nn.functional.relu(self.conv_2(x))
        5     ...
```

Figure B-18. The dropout layer in the forward function of a model

So with dropout, you have two ways of applying it, both producing similar outputs. In fact, the layer itself is an extension of the functional version of dropout, which itself is an interface. This is really up to personal preference, since both are still dropout layers and there's no real difference in behavior.

ReLU

`torch.nn.ReLU()`

ReLU, or “Rectified Linear Unit”, performs a simple activation based on the function, as shown in Figure B-19.

$$f(x) = \max(0, x)$$

Figure B-19. The general formula that ReLU follows

Here is the parameter:

- **inplace:** If set to True, it will perform the operation in place.
Default = False.

For ReLU, the graph can look like Figure B-20.

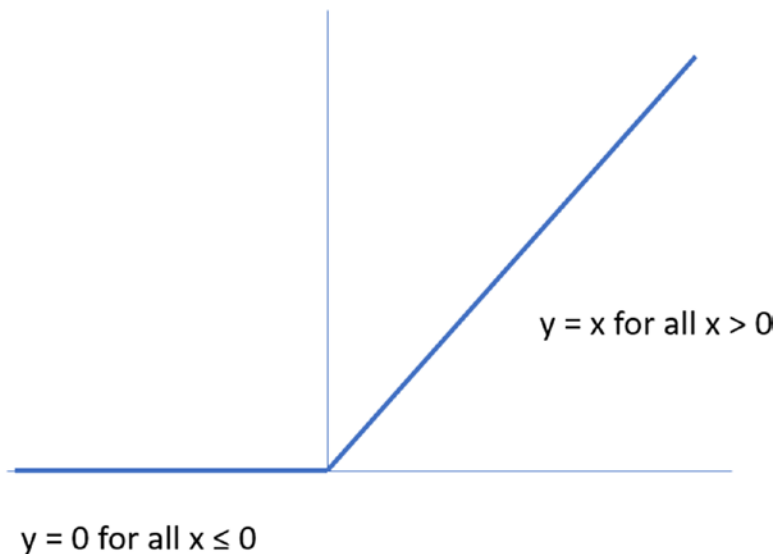


Figure B-20. The general graph of a ReLU function

Similarly to dropout, you can define this as a layer within the model itself, or apply ReLU in the forward function like so:

```
torch.nn.functional.relu(input, inplace=False)
```

Input is the previous layer.

Figure B-21 shows an example of how you can use this layer in the forward function.

Just like with dropout, you have two ways of applying ReLU, but it all boils down to personal preference.

```
In [ ]: 1 def forward(self, x):
        2     x = nn.functional.relu(self.conv_1(x))
        3     x = nn.functional.dropout(x, 0.25)
        4     x = nn.functional.relu(self.conv_2(x))
        5     ...
```

Figure B-21. The ReLU layer in the forward function of a model

Softmax

```
torch.nn.Softmax()
```

This performs a softmax on the given dimension.

The general formula for softmax is shown in Figure B-22 (K is the number of samples).

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^K e^{x_j}} \quad \text{For } i = 1, \dots, K \text{ and } x = (x_1, \dots, x_K) \in R^K$$

Figure B-22. The general formula for softmax. The parameter i goes up until the total number of samples, which is K

Here is the parameter:

- **dim:** The dimension to compute softmax along, determined by some integer **n**. This is so every slice along the dimension will sum to 1.
Default = None.

You can define this as a layer within the model itself, or apply softmax in the forward function like so:

```
torch.nn.functional.softmax(input, dim=None, _stacklevel=3)
```

Input is the previous layer.

Figure B-23 shows an example of how you can use this layer in the forward function.

```
In [ ]: 1 def forward(self, x):
        2     ...
        3     x = nn.functional.dropout(x, 0.5)
        4     x = nn.functional.softmax(self.dense_1(x), dim=1)
        5     return x
        6
```

Figure B-23. The softmax layer in the forward function of a model

However, this doesn't work well if you're using NLLL (negative log likelihood) loss, in which case you should use `log_softmax` instead.

Log_Softmax

`torch.nn.LogSoftmax()`

This performs a softmax activation on the given dimension, but passes that through a log function.

The general formula for `log_softmax` is shown in Figure B-24 (K is the number of samples).

$$\sigma(x)_i = \log\left(\frac{e^{x_i}}{\sum_{j=1}^K e^{x_j}}\right) \text{ For } i = 1, \dots, K \text{ and } x = (x_1, \dots, x_k) \in R^k$$

Figure B-24. The general formula for `log_softmax`. The value *i* goes up until the total number of samples, *K*.

Here is the parameter:

- **dim:** The dimension to compute softmax along, determined by some integer **n**. This is so every slice along the dimension will sum to 1.
Default = None.

You can define this as a layer within the model itself, or apply softmax in the forward function like so:

```
torch.nn.functional.log_softmax(input, dim=None, _stacklevel=3)
```

Input is the previous layer.

Figure B-25 shows an example of how you can use this layer in the forward function.

```
In [ ]: 1 def forward(self, x):
        2     ...
        3     x = nn.functional.dropout(x, 0.5)
        4     x = nn.functional.log_softmax(self.dense_1(x), dim=1)
        5     return x
```

Figure B-25. The log softmax layer in the forward function of a model

Sigmoid

`torch.nn.Sigmoid()`

This performs a sigmoid activation.

The sigmoid function does have its uses, primarily because it forces the input to be between 0 and 1, but it is prone to the vanishing gradient problem, and so it is seldom used in hidden layers.

There are no parameters, so it's a simple function to call.

To get an idea of what the equation is like when graphed, refer to Figure B-26.

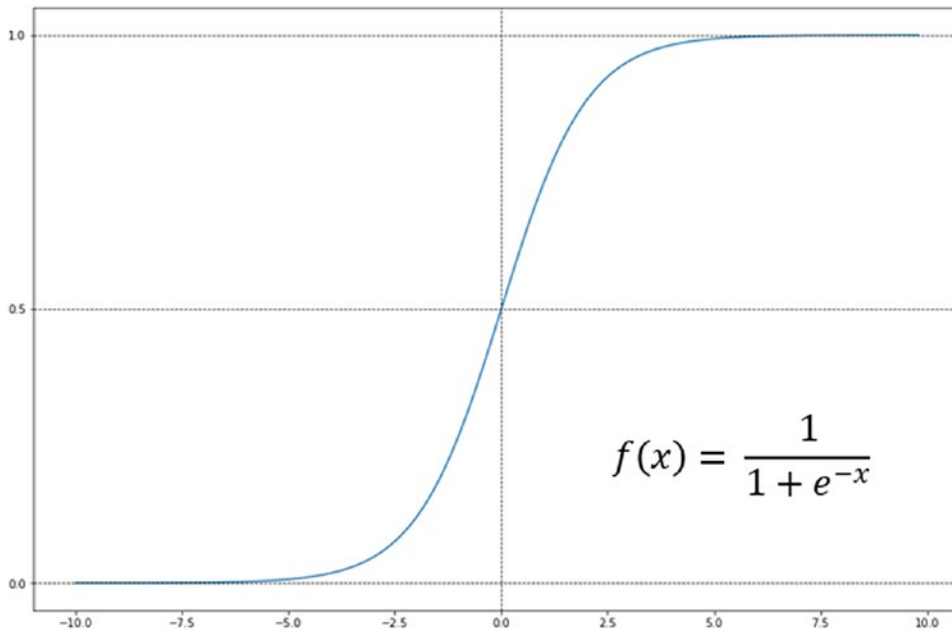


Figure B-26. The general graph of a sigmoid function

You can define this as a layer within the model itself, or apply `sigmoid` in the forward function like so:

```
torch.nn.functional.sigmoid(input)
```

Input is the previous layer.

Figure B-27 shows an example of how you can use this layer in the forward function.

```
In [ ]: 1 def forward(self, x):
        2     ...
        3     x = nn.functional.dropout(x, 0.5)
        4     x = nn.functional.sigmoid(self.dense_1(x), dim=1)
        5     return x
```

Figure B-27. The sigmoid layer in the forward function of a model

Loss Functions

MSE

```
torch.nn.MSELoss()
```

If you have questions on the notation for this equation, refer to Chapter 3. The equation is shown in Figure B-28.

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n (h_{\theta}(x^i) - y^i)^2$$

Figure B-28. The general formula for mean squared loss

Given input θ , the weights, the formula finds the average difference squared between the predicted value and the actual value. The parameter h_{θ} represents the model with the weight parameter θ passed in, so $h_{\theta}(x^i)$ would give the predicted value for x^i with model's weights θ . The parameter y^i represents the actual prediction for the data point at index i . Lastly, there are n entries in total.

This function has several parameters (two are deprecated):

- **size_average:** (Deprecated in favor of reduction.) The losses are averaged over each loss element in the batch by default (True). If set to False, then the losses are summed for each minibatch instead. Default = True.

- **reduce:** (Deprecated in favor of reduction.) The losses are averaged or summed over observations for each minibatch depending on `size_average` by default (True). If set to False, then it returns a loss per batch element and ignores `size_average`. Default = True.
- **reduction:** A string value to specify the type of reduction to be done. Choose between 'none', 'elementwise_mean', or 'sum'. 'none' means no reduction is applied, 'elementwise_mean' will divide the sum of the output by the number of elements in the output, and 'sum' will just sum the output. Default = 'elementwise_mean'. Note: specifying either `size_average` or `reduce` will override this parameter.

This loss metric can be used in autoencoders to help evaluate the difference between the reconstructed output and the original. In the case of anomaly detection, this metric can be used to separate the anomalies from the normal data points, since anomalies have a higher reconstruction error.

Cross Entropy

`torch.nn.CrossEntropyLoss()`

The equation is shown in Figure B-29.

$$J(\theta) = -\frac{1}{n} \sum_{i=0}^n y_i * \log(h_{\theta}(x_i)) + (1 - y_i) * \log(1 - h_{\theta}(x_i))$$

Figure B-29. The general formula for cross entropy loss

In this case, **n** is the number of samples in the whole data set. The parameter h_{θ} represents the model with the weight parameter θ passed in, so $h_{\theta}(x_i)$ would give the predicted value for x_i with model's weights θ . Finally, y_i represents the true labels for data point at index i . The data needs to be regularized to be between 0 and 1, so for categorical cross entropy, it must be piped through a softmax activation layer.

The categorical cross entropy loss is also called **softmax loss**.

Equivalently, you can write the previous equation as Figure B-30.

$$J(\theta) = -\frac{1}{n} \sum_{i=0}^n \sum_{j=0}^m y_{ij} * \log(h_{\theta}(x_{ij}))$$

Figure B-30. An alternate way to write the equation in Figure B-29

In this case, **m** is the number of classes.

The categorical cross entropy loss is a commonly used metric in classification tasks, especially in computer vision with convolutional neural networks.

This function has several parameters (two are deprecated):

- **weight:** (Optional) A tensor that's the size of the number of classes **n**. This is essentially a weight given to each class so that some classes are weighted more heavily in how they affect the overall loss and optimization problem.
- **size_average:** (Deprecated in favor of reduction.) The losses are averaged over each loss element in the batch by default (True). If set to False, then the losses are summed for each minibatch instead. Default = True.
- **ignore_index:** (Optional) An integer that specifies a target value that is ignored so it does not contribute to the input gradient. If size_average is True, then the loss is averaged over targets that aren't ignored.
- **reduce:** (Deprecated in favor of reduction.) The losses are averaged or summed over observations for each minibatch depending on size_average by default (True). If set to False, it returns a loss per batch element and ignores size_average. Default = True.
- **reduction:** A string value to specify the type of reduction to be done. Choose between 'none', 'elementwise_mean', or 'sum'. 'none' means no reduction is applied, 'elementwise_mean' will divide the sum of the output by the number of elements in the output, and 'sum' will just sum the output. Default = 'elementwise_mean'. Note: Specifying either size_average or reduce will override this parameter.

Optimizers

SGD

```
torch.optim.SGD()
```

This is the **stochastic gradient descent** optimizer, a type of algorithm that aids in the backpropagation process by adjust the weights. It is commonly used as a training algorithm in a variety of machine learning applications, including neural networks.

This function has several parameters:

- **params**: Some iterable of parameters to optimize, or dictionaries with parameter groups. This can be something like `model.parameters()`.
- **lr**: A float value specifying the learning rate.
- **momentum**: (Optional) Some float value specifying the momentum factor. This parameter helps accelerate the optimization steps in the direction of the optimization, and helps reduce oscillations when the local minimum is overshoot (refer to Chapter 3 to refresh your understanding on how a loss function is optimized). Default = 0.
- **weight_decay**: A l2_penalty for weights that are too high, helping incentivize smaller model weights. Default = 0.
- **dampening**: The dampening factor for momentum. Default = 0.
- **nesterov**: A Boolean value to determine whether or not to apply Nesterov momentum. Nesterov momentum is a variation of momentum where the gradient is computed not from the current position, but from a position that takes into account the momentum. This is because the gradient always points in the right direction, but the momentum might carry the position too far forward and overshoot. Since this doesn't use the current position but instead some intermediate position that takes into account momentum, the gradient from that position can help correct the current course so that the momentum doesn't carry the new weights too far forward. It essentially helps for more accurate weight updates and helps converge faster. Default = False.

Adam

```
torch.optim.Adam()
```

The Adam optimizer is an algorithm that extends upon SGD. It has grown quite popular in deep learning applications in computer vision and in natural language processing.

This function has several parameters:

- **params:** Some iterable of parameters to optimize, or dictionaries with parameter groups. This can be something like `model.parameters()`.
- **lr:** A float value specifying the learning rate. Default = 0.001 (or 1e-3).
- **betas:** (Optional) A tuple of two floats to define the beta values `beta_1` and `beta_2`. The paper describes good results with (0.9, 0.999) respectively, which is also the default value.
- **eps:** (Optional). Some float value where $\epsilon \geq 0$. Epsilon is some small number, described as 10E-8 in the paper, to help prevent division by 0. Default is 1e-8.
- **weight_decay:** A l2_penalty for weights that are too high, helping incentivize smaller model weights. Default = 0.
- **amsgrad:** A Boolean on whether or not to apply the AMSGrad version of this algorithm. For more details on the implementation of this algorithm, check out “On the Convergence of Adam and Beyond.” Default=False.

RMSProp

```
torch.optim.RMSprop()
```

RMSprop is a good algorithm for recurrent neural networks. RMSprop is a gradient-based optimization technique developed to help address the problem of gradients becoming too large or too small. RMSprop helps combat this problem by normalizing the gradient itself using the average of the squared gradients. In Chapter 7, it’s explained that one of the problems with RNNs is the vanishing/exploding gradient problem,

leading to the development of LSTMs and GRU networks. And so it's of no surprise that RMSprop pairs well with recurrent neural networks.

This function has several parameters:

- **params:** Some iterable of parameters to optimize, or dictionaries with parameter groups. This can be something like `model.parameters()`.
- **lr:** A float value specifying the learning rate. Default = 0.01 (or 1e-2).
- **momentum:** (Optional). Some float value specifying the momentum factor. This parameter helps accelerate the optimization steps in the direction of the optimization, and helps reduce oscillations when the local minimum is overshoot (refer to Chapter 3 to refresh your understanding on how a loss function is optimized). Default = 0.
- **alpha:** (Optional) A smoothing constant. Default = 0.99
- **eps:** (Optional). Some float value where $\epsilon \geq 0$. Epsilon is some small number, described as 10E-8 in the paper, to help prevent division by 0. Default is 1e-8.
- **centered:** (Optional) If True, then compute the centered RMSprop and have the gradient normalized by an estimation of its variance. Default = False.
- **weight_decay:** An l2_penalty for weights that are too high, helping incentivize smaller model weights. Default = 0.

Hopefully by now you understand how PyTorch works by looking at some of the functionality that it offers. You built and applied a model to the MNIST data set in an organized format, and you looked at some of the basics of PyTorch by learning about the layers, how models are constructed, how activations are performed, and what the loss functions and optimizers are.

Temporal Convolutional Network in PyTorch

Now, you will look at an example of using PyTorch to construct a temporal convolutional network and apply it to the credit card data set from Chapter 7.

Dilated Temporal Convolutional Network

The particular TCN you will reconstruct in PyTorch is the dilated TCN in Chapter 7.

Once again, you begin with your imports and define your device (Figure B-31 and Figure B-32).

```
import numpy as np
import pandas as pd
import keras
from sklearn.model_selection import train_test_split
from sklearn.preprocessing.data import StandardScaler
import torch
import torch.nn as nn
import torch.nn.functional as F

# Hyperparameters
num_epochs = 30
num_classes = 2
learning_rate = 0.002

device = torch.device('cuda:0' if torch.cuda.is_available() else
'cpu')
```

Figure B-31. *Importing the necessary modules*

```
In [14]: 1 import numpy as np
          2 import pandas as pd
          3 from sklearn.model_selection import train_test_split
          4 from sklearn.preprocessing.data import StandardScaler
          5 import torch
          6 import torch.nn as nn
          7 import torch.nn.functional as F
          8
          9 # Hyperparameters
         10 num_epochs = 30
         11 num_classes = 2
         12 learning_rate = 0.002
         13
         14 device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

Figure B-32. *The code in Figure B-31 in a Jupyter cell*

Next, you load your data set (Figure B-33).

```
df = pd.read_csv("datasets/creditcardfraud/creditcard.csv",
sep=",", index_col=None)

print(df.shape)

df.head()
```

Figure B-33. Loading your data set and displaying the first five rows

The output should look somewhat like Figure B-34.

```
In [2]: 1 df = pd.read_csv("datasets/creditcardfraud/creditcard.csv", sep=",", index_col=None)
        2 print(df.shape)
        3 df.head()

(284807, 31)

Out[2]:
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V31
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.01831
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.22571
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.24791
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.10831
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.00941

5 rows × 31 columns

Figure B-34. The output of the code in Figure B-33

You need to standardize the values for Time and for Amount since they can get large. Everything else has already been standardized in the data set. Run the code in Figure B-35.

```
df['Amount'] =
StandardScaler().fit_transform(df['Amount'].values.reshape(-1, 1))

df['Time'] =
StandardScaler().fit_transform(df['Time'].values.reshape(-1, 1))

df.tail()
```

Figure B-35. Standardizing the values in the columns Amount and Time

The output should look somewhat like Figure B-36.

```
In [3]: 1 df['Amount'] = StandardScaler().fit_transform(df['Amount'].values.reshape(-1, 1))
        2 df['Time'] = StandardScaler().fit_transform(df['Time'].values.reshape(-1, 1))
        3 df.tail()
```

Out[3]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	
284802	1.641931	-11.881118	10.071785	-9.834783	-2.066856	-5.364473	-2.606837	-4.918215	7.305334	1.91...
284803	1.641952	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294869	0.58...
284804	1.641974	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.708417	0.43...
284805	1.641974	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.679145	0.39...
284806	1.642058	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.414650	0.48...

5 rows × 31 columns

<

Figure B-36. The output of the code in figure B-35.

Now you define your normal and anomaly data sets (see Figure B-37.

```
anomalies = df[df["Class"] == 1]
normal = df[df["Class"] == 0]

anomalies.shape, normal.shape
```

Figure B-37. Defining the anomaly and normal data sets

The output should look like Figure B-38.

```
In [4]: 1 anomalies = df[df["Class"] == 1]
        2 normal = df[df["Class"] == 0]
        3
        4 anomalies.shape, normal.shape
        5
```

Out[4]: ((492, 31), (284315, 31))

Figure B-38. The output of the code in Figure B-37

After isolating the anomalies from the normal data, let's create your training and testing sets (see Figure B-39).

```
for f in range(0, 20):
    normal = normal.iloc[np.random.permutation(len(normal))]

data_set = pd.concat([normal[:10000], anomalies])

x_train, x_test = train_test_split(data_set, test_size = 0.4,
    random_state = 42)

x_train = x_train.sort_values(by=['Time'])
x_test = x_test.sort_values(by=['Time'])

y_train = x_train["Class"]
y_test = x_test["Class"]

x_train.head(10)
```

Figure B-39. *The creation of the training and testing data sets*

The output should look somewhat like Figure B-40.

```
In [5]: 1 for f in range(0, 20):
        2     normal = normal.iloc[np.random.permutation(len(normal))]
        3
        4
        5 data_set = pd.concat([normal[:10000], anomalies])
        6
        7 x_train, x_test = train_test_split(data_set, test_size = 0.4, random_state = 42)
        8
        9 x_train = x_train.sort_values(by=['Time'])
       10 x_test = x_test.sort_values(by=['Time'])
       11
       12 y_train = x_train["Class"]
       13 y_test = x_test["Class"]
       14
       15 x_train.head(10)
```

Out[5]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	
8	-1.996436	-0.894286	0.286157	-0.113192	-0.271526	2.669599	3.721818	0.370145	0.851084	-0.392
99	-1.995151	1.232996	0.189454	0.491040	0.633673	-0.511574	-0.990609	0.066240	-0.196940	0.075
118	-1.994983	-0.997176	0.228365	1.715340	-0.420067	0.560838	0.564725	0.846047	0.197491	-0.097
177	-1.994182	1.194066	-0.072582	0.635286	0.768616	-0.735584	-0.673466	-0.146299	-0.065653	0.646
225	-1.993488	-2.687978	4.390230	-2.360483	0.360829	1.310192	-1.645253	2.327776	-1.727825	4.324
259	-1.992729	0.726749	-0.528042	0.050366	1.373621	-0.124122	0.415688	0.259555	0.085114	-0.003
356	-1.991087	1.260328	0.299161	0.527681	0.614899	-0.420592	-0.977533	0.108485	-0.244502	-0.058
374	-1.990834	1.124355	-0.132953	0.588467	0.804871	-0.726266	-0.521875	-0.167010	0.059298	0.366
379	-1.990729	-1.092301	0.430750	1.249785	0.429757	1.272076	0.548203	-0.120592	0.452571	-0.414
400	-1.990476	-0.695818	0.581773	2.378180	0.063396	0.329119	-0.449865	1.269104	-0.758363	0.381

10 rows × 10 columns

< 

Figure B-40. The output of the code in Figure B-39

After defining your data sets, you need to reshape the values so that your neural network can accept them (see Figure B-41).

```
x_train = np.array(x_train).reshape(x_train.shape[0], 1,
x_train.shape[1])

x_test = np.array(x_test).reshape(x_test.shape[0], 1,
x_test.shape[1])

y_train = np.array(y_train).reshape(y_train.shape[0] , 1)
y_test = np.array(y_test).reshape(y_test.shape[0], 1)

print("Shapes:\nx_train:%s\ny_train:%s\n" % (x_train.shape,
y_train.shape))

print("x_test:%s\ny_test:%s\n" % (x_test.shape,
y_test.shape))
```

Figure B-41. Reshaping the training and testing data sets so you can pass them into the model

The output should look like Figure B-42.

```
In [6]: 1 x_train = np.array(x_train).reshape(x_train.shape[0], 1, x_train.shape[1])
2 x_test = np.array(x_test).reshape(x_test.shape[0], 1, x_test.shape[1])
3
4 y_train = np.array(y_train).reshape(y_train.shape[0] , 1)
5 y_test = np.array(y_test).reshape(y_test.shape[0], 1)
6
7 print("Shapes:\nx_train:%s\ny_train:%s\n" % (x_train.shape, y_train.shape))
8 print("x_test:%s\ny_test:%s\n" % (x_test.shape, y_test.shape))

Shapes:
x_train:(6295, 1, 31)
y_train:(6295, 1)

x_test:(4197, 1, 31)
y_test:(4197, 1)
```

Figure B-42. The output of the code in Figure B-41

Now you can define your model (Figure B-43 and Figure B-44).

```
class TCN(nn.Module):
    def __init__(self):
        super(TCN, self).__init__()

        self.conv_1 = nn.Conv1d(1, 128, kernel_size=2, dilation=1,
padding=((2-1) * 1))

        self.conv_2 = nn.Conv1d(128, 128, kernel_size=2, dilation=2,
padding=((2-1) * 2))

        self.conv_3 = nn.Conv1d(128, 128, kernel_size=2, dilation=4,
padding=((2-1) * 4))

        self.conv_4 = nn.Conv1d(128, 128, kernel_size=2, dilation=8,
padding=((2-1) * 8))

        self.dense_1 = nn.Linear(31*128, 128)
        self.dense_2 = nn.Linear(128, num_classes)
```

Figure B-43. The first part of the TCN class

```

def forward(self, x):
    x = self.conv_1(x)
    x = x[:, :, :-self.conv_1.padding[0]]
    x = F.relu(x)
    x = F.dropout(x, 0.05)
    x = self.conv_2(x)
    x = x[:, :, :-self.conv_2.padding[0]]
    x = F.relu(x)
    x = F.dropout(x, 0.05)
    x = self.conv_3(x)
    x = x[:, :, :-self.conv_3.padding[0]]
    x = F.relu(x)
    x = F.dropout(x, 0.05)
    x = self.conv_4(x)
    x = x[:, :, :-self.conv_4.padding[0]]
    x = F.relu(x)
    x = F.dropout(x, 0.05)
    x = x.view(-1, 31*128)
    x = F.relu(self.dense_1(x))
    x = self.dense_2(x)
    return F.log_softmax(x, dim=1)

```

Figure B-44. *The forward function in the TCN class*

The code for the model should look like [Figure B-45](#).

```

In [13]: 1 class TCN(nn.Module):
2         def __init__(self):
3             super(TCN, self).__init__()
4
5             self.conv_1 = nn.Conv1d(1, 128, kernel_size=2, dilation=1, padding=((2-1) * 1))
6             self.conv_2 = nn.Conv1d(128, 128, kernel_size=2, dilation=2, padding=((2-1) * 2))
7             self.conv_3 = nn.Conv1d(128, 128, kernel_size=2, dilation=4, padding=((2-1) * 4))
8             self.conv_4 = nn.Conv1d(128, 128, kernel_size=2, dilation=8, padding=((2-1) * 8))
9             self.dense_1 = nn.Linear(31*128, 128)
10            self.dense_2 = nn.Linear(128, num_classes)
11
12        def forward(self, x):
13            x = self.conv_1(x)
14            x = x[:, :, :-self.conv_1.padding[0]]
15            x = F.relu(x)
16            x = F.dropout(x, 0.05)
17            x = self.conv_2(x)
18            x = x[:, :, :-self.conv_2.padding[0]]
19            x = F.relu(x)
20            x = F.dropout(x, 0.05)
21            x = self.conv_3(x)
22            x = x[:, :, :-self.conv_3.padding[0]]
23            x = F.relu(x)
24            x = F.dropout(x, 0.05)
25            x = self.conv_4(x)
26            x = x[:, :, :-self.conv_4.padding[0]]
27            x = F.relu(x)
28            x = F.dropout(x, 0.05)
29            x = x.view(-1, 31*128)
30            x = F.relu(self.dense_1(x))
31            x = self.dense_2(x)
32            return F.log_softmax(x, dim=1)

```

Figure B-45. The code from Figures B-43 and B-44 in a Jupyter cell. This defines the entire model

Now you can define your training and testing functions (Figure B-46, Figure B-48, and Figure B-49).


```

def train(model, device, x_train, y_train, criterion, optimizer,
epoch, save_dir='TCN_CreditCard_PyTorch.ckpt'):
    total_step = len(x_train)

    x_train = torch.Tensor(x_train).cuda().float()
    y_train = torch.Tensor(y_train).cuda().long()

    x_train.to(device)
    y_train.to(device)

    # Forward pass
    outputs = model(x_train)
    loss = criterion(outputs, y_train.squeeze(1))

    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    print('Epoch {}/{}'.format(epoch+1,
num_epochs, loss.item()))

    torch.save(model.state_dict(), save_dir)

```

Figure B-46. The training function. Since you don't have data loaders, you pass in `x_train` and `y_train` directly into the GPU after converting them to tensors. The inputs then pass through, and the gradients are calculated.

```

In [12]: 1 def train(model, device, x_train, y_train, criterion, optimizer, epoch, save_dir='TCN_CreditCard_PyTorch.ckpt'):
          2     total_step = len(x_train)
          3
          4     x_train = torch.Tensor(x_train).cuda().float()
          5     y_train = torch.Tensor(y_train).cuda().long()
          6
          7     x_train.to(device)
          8     y_train.to(device)
          9
         10     # Forward pass
         11     outputs = model(x_train)
         12     loss = criterion(outputs, y_train.squeeze(1))
         13
         14     # Backward and optimize
         15     optimizer.zero_grad()
         16     loss.backward()
         17     optimizer.step()
         18
         19     print('Epoch {}/ {}, Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))
         20
         21     torch.save(model.state_dict(), save_dir)

```

Figure B-47. The code from Figure B-46 in a Jupyter cell

```

from sklearn.metrics import roc_auc_score

def test(model, device, x_test, y_test):
    preds = []
    y_true = []

    # Set model to evaluation mode.
    model.eval()

    with torch.no_grad():
        correct = 0
        total = 0

        x_test = torch.Tensor(x_test).cuda().float()
        y_test = torch.Tensor(y_test).cuda().long()

        x_test = x_test.to(device)
        y_test = y_test.to(device)
        y_test = y_test.squeeze(1)
        outputs = model(x_test)
        _, predicted = torch.max(outputs.data, 1)
        total += y_test.size(0)
        correct += (predicted == y_test).sum().item()
        detached_pred = predicted.detach().cpu().numpy()
        detached_label = y_test.detach().cpu().numpy()

```

Figure B-48. The testing function. Since there are no data loaders, the testing sets must be converted into a tensor and passed into a GPU before being able to make predictions on them. The AUC score is then generated along with an accuracy value

The rest of the testing function code is shown in [Figure B-49](#).

```

for f in range(0, len(detached_label)):
    preds.append(detached_pred[f])
    y_true.append(detached_label[f])

print('Test Accuracy of the model on the 10000
test images: {:.2%}'.format(correct / total))

preds = np.eye(num_classes)[preds]
y_true = np.eye(num_classes)[y_true]
auc = roc_auc_score(np.round(preds), y_true)
print("AUC: {:.2%}".format (auc))

```

Figure B-49. The rest of the testing function. This deals with calculating the AUC score and accuracy value

The entire test function should look like Figure B-50.

```

In [218]: 1 from sklearn.metrics import roc_auc_score
2
3 def test(model, device, x_test, y_test):
4     preds = []
5     y_true = []
6
7     # Set model to evaluation mode.
8     model.eval()
9     with torch.no_grad():
10         correct = 0
11         total = 0
12
13         x_test = torch.Tensor(x_test).cuda().float()
14         y_test = torch.Tensor(y_test).cuda().long()
15
16         x_test = x_test.to(device)
17         y_test = y_test.to(device)
18         y_test = y_test.squeeze(1)
19         outputs = model(x_test)
20         _, predicted = torch.max(outputs.data, 1)
21         total += y_test.size(0)
22         correct += (predicted == y_test).sum().item()
23         detached_pred = predicted.detach().cpu().numpy()
24         detached_label = y_test.detach().cpu().numpy()
25         for f in range(0, len(detached_label)):
26             preds.append(detached_pred[f])
27             y_true.append(detached_label[f])
28
29     print('Test Accuracy of the model on the 10000 test images: {:.2%}'.format(correct / total))
30
31     preds = np.eye(num_classes)[preds]
32     y_true = np.eye(num_classes)[y_true]
33     auc = roc_auc_score(np.round(preds), y_true)
34     print("AUC: {:.2%}".format (auc))

```

Figure B-50. The entire test function, comprised of code from Figures B-48 and B-49

Finally, you can train our model as shown in Figure B-51.

```
model = TCN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),
                               lr=learning_rate)

## Training phase

for epoch in range(0, num_epochs):
    train(model, device, x_train, y_train, criterion,
          optimizer, epoch)
```

Figure B-51. *Initializing the TCN model, defining the criterion as the cross entropy loss, and defining the optimizer (Adam optimizer)*

The output should look somewhat like Figure B-52.

```
In [15]: 1 model = TCN().to(device)
2 criterion = nn.CrossEntropyLoss()
3 optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
4
5
6
7 ## Training phase
8
9 for epoch in range(0, num_epochs):
10     train(model, device, x_train, y_train, criterion, optimizer, epoch)
11
Epoch 1/30, Loss: 0.7319
Epoch 2/30, Loss: 0.5128
Epoch 3/30, Loss: 0.3454
Epoch 4/30, Loss: 0.2430
Epoch 5/30, Loss: 0.1245
Epoch 6/30, Loss: 0.1041
Epoch 7/30, Loss: 0.0684
Epoch 8/30, Loss: 0.0737
Epoch 9/30, Loss: 0.0646
Epoch 10/30, Loss: 0.0672
Epoch 11/30, Loss: 0.0650
Epoch 12/30, Loss: 0.0545
Epoch 13/30, Loss: 0.0526
Epoch 14/30, Loss: 0.0428
Epoch 15/30, Loss: 0.0403
Epoch 16/30, Loss: 0.0409
Epoch 17/30, Loss: 0.0453
Epoch 18/30, Loss: 0.0366
Epoch 19/30, Loss: 0.0357
Epoch 20/30, Loss: 0.0358
Epoch 21/30, Loss: 0.0351
Epoch 22/30, Loss: 0.0347
Epoch 23/30, Loss: 0.0342
Epoch 24/30, Loss: 0.0333
Epoch 25/30, Loss: 0.0329
Epoch 26/30, Loss: 0.0322
Epoch 27/30, Loss: 0.0316
Epoch 28/30, Loss: 0.0314
Epoch 29/30, Loss: 0.0307
Epoch 30/30, Loss: 0.0304
```

Figure B-52. The output of the training process

And now you can evaluate your model (see Figure B-53).

```
## Testing phase

test(model, device, x_test,
      y_test)
```

Figure B-53. Calling the test function

The output should look somewhat like Figure B-54.

```
In [16]: 1  ## Testing phase
          2
          3  test(model, device, x_test, y_test)

Test Accuracy of the model on the 10000 test images: 99.14%
AUC: 98.98%
```

Figure B-54. The output AUC value of the testing function

With the end of this example, you will have created a TCN in both Keras and PyTorch. This way, you'll have a good way to compare how the model is built, trained, and evaluated in both frameworks, allowing you to observe the similarities and differences in how both frameworks handle those processes.

By now, you should have a better understanding of how PyTorch works, especially with how it's meant to be more intuitive. Think back to the training function and the process of converting the data sets, passing them through the GPU and through the model, calculated the gradients, and backpropagating. Though it's not abstracted away from you like in Keras, it still makes logical sense in that the functions called directly correlate to the training process of a neural network.

Summary

PyTorch is a low-level tool that allows you to quickly create, train, and test your own deep learning models, although it is more complicated than doing the same in Keras. However, it offers you much more functionality, flexibility, and customizability compared to Keras, and compared to TensorFlow, it is much lighter on syntax. With PyTorch, you don't have to worry about switching frameworks as you get more advanced because of the functionality that it offers, making it a great tool to use when conduct deep learning research. PyTorch should be enough for most of your needs as you become more experienced with deep learning, and using either PyTorch or TensorFlow (or tf.keras + TensorFlow) is just a matter of personal preference.

Index

A

Adam optimizer, 103, 346, 391

Anomaly detection

abnormal behavior, 299

data points

location, 6

range of density/tensile

strength, 5, 6

sample screw falls, 7, 8

defined, 298

example, 298, 299

taxi cabs, number of pickups, 11–15

time series, 9, 11

uses

data breaches, 20

identity theft, 21

manufacturing, 21, 22

medicine, 22, 23

networking, 22

Arctanh function, 361

Area under the curve of the

receiver operating

characteristic (AUROC), 29

Autoencoders, 302

activation functions, 131

anomalies, 140

CNN

compile model, 147, 148

neural network, 145, 146

display encoded images, 148, 149

import packages, 144, 145

load MNIST data, 145

training process, 150–152

compile model, 132

confusion matrix, 139

deep neural network, 142, 143

importing packages, 127, 128

latent/compressed representation, 125

measure anomalies, 137

neural network, 123, 124

Pandas dataframe, 129, 130

precision/recall code, 138

reconstruction loss, 126

splitting data, 131

training process, 132, 134–136

visualize results via confusion

matrix, 128, 129

B

Banking sector

autoencoders, 302

credit card, 302

Bi-directional encoders, 257

Boltzmann machine

bidirectional neural network, 179

derivations, 180

generative network, 180

graph, 180

C

- categorical() function, 275
- Confusion matrix, 26, 27, 139
- Context-based anomalies, 16
- Contrastive divergence (CD), 186
- Convolutional neural
 - networks (CNN), 85, 144, 304
- Credit card data set
 - AUC scores, 193
 - free energy *vs.* probabilities, 195, 196
 - modules, import, 187
 - normal data points, 193, 194
 - output training model, 192
 - parameters, 191, 192
 - RBM model, 190
 - standardized values, 189
 - training process, 188
 - training/testing sets, 190
- Cybersecurity
 - DOS attack, 310
 - intrusion activity, 311
 - TCP connections, 311
 - Trojan, 310

D

- Data point-based anomalies, 16
- Data science
 - accuracy, 28
 - AUC score, 32, 33
 - AUROC, 29
 - confusion matrix, 26
 - definitions, 26
 - F1 score, 29
 - precision, 28
 - recall, 28
 - ROC curve, 29
 - training data set, 30

- type I error, 26
 - type II error, 26
- Deep belief networks (DBN), 180
- Deep Boltzmann
 - machines (DBM), 180
- Deep learning, 309
 - artificial neural networks
 - activation function, 76
 - backpropagation, 81, 83
 - cost function, 82
 - gradient, 82
 - hidden layer, 77, 80
 - input layer, 78, 79
 - Keras framework, 84
 - layers, 76
 - learning rate, 83
 - mean squared error, 82
 - neuron, 74–76
 - output layer, 81
 - PyTorch, 84
 - tensorflow, 84
 - GPU, 73
 - models, 73
- Deep learning-based anomaly
 - detection
 - challenges, 317
 - key steps, 316, 317
- Denial of service (DOS) attack, 310
- Denoising autoencoder
 - compiling model, 157
 - depiction, 153
 - display encoded images, 159
 - evaluate model, 158
 - import packages, 154
 - load MNIST images, 154
 - load/reshape images code, 155
 - neural network, 155, 156
 - training process, 158, 160–162

Dilated TCN, anomaly detection

- AUC score, [281, 282](#)
- classification report, [282](#)
- confusion matrix, [282](#)
- data frame, [270, 271, 273](#)
- import packages, [267, 268](#)
- model, defined, [276, 277](#)
- model summary, [278, 279](#)
- shape data sets, [274–276](#)
- sort by Time column, [274](#)
- standardize values, [271, 272](#)
- training process, [280](#)
- visualization class, [268, 269](#)

Dilated temporal convolutional

- network (TCN)
 - acausal network, [266](#)
 - anomaly detection (*see* Dilated TCN, anomaly detection)
 - causal network, [266, 267](#)
 - dilation factor, [262, 263](#)
 - feature map, [264](#)
 - filter weights, [264](#)
 - input vector, [264](#)
 - one-dimensional
 - convolutions, [264](#)
 - output vector, [265](#)
- dilation factor, [262, 263](#)

E

ED-TCN, anomaly detection

- AUC score, [294, 295](#)
- decoding stage, [290](#)
- encoding stage, [289](#)
- evaluate performance, [294](#)
- import modules, [286, 287](#)
- model summary, [292](#)
- reshape data sets, [288](#)

- train, data, [293](#)

- zero padding, [292, 293](#)

Encoder-decoder TCN

- anomaly detection (*see* ED-TCN, anomaly detection)
- decoding stage, [285](#)
- encoding stage, [284](#)
- model structure, [283, 284](#)
- upsampling, [285, 286](#)

Environmental use case

- air quality index, [303, 304](#)
- deforestation, [303](#)

Epoch, [86](#)**F**

- Filter/kernel operation, [378, 379](#)

- Finance and insurance industries, [308, 309](#)

G

- Gradient-based optimization
 - technique, [347, 391](#)

H

- Healthcare, [304–306](#)

I, J

- inception-v3 model, [259](#)

Isolation forest

- mutant fish, [34, 35](#)
- works
 - calculate AUC, [49](#)
 - categorical values, [44](#)
 - data sets, [39, 40](#)
 - filtering df, [41](#)

INDEX

Isolation forest (*cont.*)

- histogram, [50, 51](#)
- KDDCUP 1999 data set, [36, 37](#)
- label encoder, [42–44](#)
- Matplotlib, [38](#)
- numpy module, [37, 38](#)
- Pandas module, [38](#)
- parameters, [47](#)
- scikit-learn, [38](#)
- training set, [45](#)
- validation set, [46](#)

K

KDDCUP data set

- anomalies *vs.* normal data points, [211](#)
- anomalous data, [201, 210](#)
- AUC scores, [203, 209](#)
- define column, [198](#)
- exploding gradient, [205](#)
- free energy *vs.* probability, [211](#)
- HTTP attacks, [199](#)
- Jupyter cell, [204](#)
- label encoder, [199, 200, 204](#)
- modules, import, [197](#)
- output, [201–203, 206](#)
- training output, [207, 208](#)
- training/testing sets, [205](#)
- unsupervised training, [206](#)

Keras, [84](#)

- activation function, [331](#)
- activation map/feature map, [95](#)
- adam optimizer, [346, 347](#)
- AUC score, [107, 109](#)
- back end (TensorFlow operations), [358, 359](#)
- binary accuracy, [343, 344](#)
- categorical accuracy, [344, 345](#)

CNN, [85](#)

- compiling model, [94](#)
- data set, [87](#)
- deep learning model, [319](#)
- dense layer, [102, 329, 330](#)
- dropout layer, [101, 331, 332](#)
- epoch, [86](#)
- evaluate function, [327](#)
- file path, [328](#)
- filter, [96–99](#)
- flatten layer, [332, 333](#)
- functional model, [321](#)
- image properties, [89, 90](#)
- input layer, [328, 329](#)
- matplotlib, [86](#)
- Max pooling, [100, 339–340](#)
- min-max normalization, [90–92](#)
- MNIST dataset, [85](#)
- ModelCheckpoint, [351, 352](#)
- model compilation/training
 - ModelCheckpoint(), [323](#)
 - model.fit() function, [324](#)
 - parameters, [322, 323](#)
 - verbosity, [324, 325](#)
- model evaluation/prediction, [326](#)
- normalization/feature scaling, [90](#)
- one-dimensional convolutional
 - layer, [334, 335](#)
- parameters, [326](#)
- pooling layer, [101](#)
- prediction function, [327](#)
- ReLU function, [102, 103](#)
- RLU, [349, 350](#)
- RMSprop, [347, 348](#)
- sequential model, [95, 321](#)
- sigmoid activation, [350, 351](#)
- softmax activation, [348](#)
- Spatial Dropout, [333, 334](#)

- standardization, [91](#)
- TensorBoard (*see* TensorBoard)
- TensorFlow/PyTorch, [319](#), [320](#)
- training data, [105](#)
- transformed data, [93](#)
- 2D convolutional layer, [336](#), [337](#)
- Unit length scaling, [91](#)
- vector representation, [92](#), [93](#)
- ZeroPadding, [338](#), [339](#)

Kernel trick, [61](#)

L

- Label encoder, [42–44](#)
- Long Short-Term Memory (LSTM) models
 - activation function, [219](#), [220](#)
 - anomalies, [242](#)
 - anomaly detection
 - adam optimizer, [230](#)
 - dataframe, [230](#)
 - dataset, [226](#)
 - errors, [224](#)
 - import packages, [223](#), [224](#)
 - plotting time series, [227](#)
 - value column, [228](#), [229](#)
 - visualize errors, [225](#)
 - arguments, [231](#), [232](#)
 - compute threshold, [240](#), [241](#)
 - dataframe, [242](#)
 - definition, [218](#)
 - linear/non-linear data plots, [220](#)
 - RNN, [216](#), [217](#)
 - sequence/time series, [213–215](#)
 - sigmoid activation function, [221](#), [222](#)
 - tanh function, [219](#)
 - testing dataset, [239](#)
 - time series, examples
 - ambient_temperature_system_
 - failure, [251–254](#)

- art_daily_jumpsdown, [246–248](#)
- art_daily_nojump, [244–246](#)
- art_daily_no_noise, [243](#), [244](#)
- art_daily_perfect_square_
 - wave, [248–250](#)
- art_load_balancer_spikes, [250](#), [251](#)
- rds_cpu_utilization, [254](#), [255](#)
- training process, [235–238](#)

Loss functions

- cross entropy loss, [388](#), [389](#)

Keras

- categorical cross entropy, [341](#)
- mean squared error, [340](#), [341](#)
- sparse categorical cross
 - entropy, [342](#), [343](#)

- MSE, [387](#), [388](#)

M

Manufacturing sector

- automation, [313](#)
- sensors, [313](#), [314](#)

Matplotlib, [38](#)

Mean normalization, [91](#)

Mean squared error, [82](#), [230](#)

Mean squared loss (MSE), [387](#)

Modified National Institute of

- Standards and Technology
 - (MNIST), [85](#), [392](#)

Momentum, [187](#), [346](#)

N

Nesterov momentum, [346](#), [390](#)

Noise removal, [18](#)

Normalization/feature

- scaling, [90](#)

novelties.head(), [202](#)

Novelty detection, [18](#), [19](#), [51](#)

O

One class SVM

- data points, [58, 59](#)
- gamma, [61](#)
- hyperplane, [54–57](#)
- kernel, [61](#)
- novelties, [62](#)
- regularization, [61](#)
- semi-supervised anomaly detection, [51](#)
- support vector, [54](#)
- visualize data, [52, 53](#)
- works
 - accuracy, [67–69](#)
 - AUC score, [69, 70](#)
 - categorical values, [64](#)
 - data points, [71](#)
 - data sets shapes, [65, 66](#)
 - filtering data, [64](#)
 - importing modules, [63](#)
 - KDDCUP 1999 data set, [63](#)
 - label encoder, [65](#)
 - model, [66](#)

Optimizers

- adam, [391](#)
- RMSprop, [391, 392](#)
- SGD, [390](#)

Outlier detection, [18](#)**P, Q**Pattern-based anomalies, [17](#)Persistent contrastive divergence (PCD), [186](#)Probability function, [183, 184](#)PyTorch, [84](#)

- AUC score, [119, 121](#)
- compatibility, [362](#)
- creating CNN, [115–117](#)
- creating model, [114](#)

deep learning library, [361](#)hyperparameters, [112, 371, 372](#)Jupyter cell, [365, 367, 369, 371, 374](#)

layers

- Conv1d, [377, 378](#)
- Conv2d, [378, 379](#)
- dropout, [382](#)
- linear, [380](#)
- log_softmax, [385, 386](#)
- MaxPooling1D, [380](#)
- MaxPooling2D, [381](#)
- ReLU, [383, 384](#)
- sigmoid function, [386, 387](#)
- softmax, [384, 385](#)
- ZeroPadding2D, [381](#)

loss functions, [387](#)low-level language, [361](#)model, [366](#)network creation, [365](#)optimizer, [373](#)sequential *vs.* modulelist, [376, 377](#)temporal convolutional network, [393](#)TensorFlow, [361](#)tensor operations, [362–364](#)testing, [369–370](#)

training

- algorithm, [368](#)
- data, [118](#)
- function, [369](#)
- process, [119, 375](#)

training/testing data sets, [113](#)**R**Receiver operating characteristic (ROC)
curve, [29](#)Rectified Linear Unit (ReLU), [102, 131, 349, 350](#)

Recurrent neural network (RNN), 216, 257
 Restricted boltzmann
 machine (RBM), 180
 credit card data set (*see* Credit card data set)
 energy function, 182
 expected value, 186
 KDDCUP data set (*see* KDDCUP data set)
 layers, 181
 probability function, 183, 184
 sigmoid function, 185
 unsupervised learning algorithm, 186
 vector *vs.* transposed vector, 183
 visual representation, 181, 182
 Retail industry, 315, 316

S

Scikit-learn, 38, 42
 Semi-supervised anomaly
 detection, 19, 51
 Smart home system, 315
 Social media platforms, 307, 308
 Softmax, 131
 Sparse autoencoders, 140–142
 Standardization (z-score
 normalization), 91
 Stochastic gradient descent (SGD), 345,
 346, 390
 Supervised anomaly detection,
 19, 262, 283
 Support vector machine (SVM), 53, 61

T

tanh function, 219, 222
 Telecom sector

 roaming, 300
 service disruption, 300, 301
 TCN or LSTM algorithms, 301
 Temporal convolutional
 networks (TCNs)
 advantages, 258
 anomaly/normal data, 395
 data set, 394
 defined, 257
 disadvantages, 258, 259
 import modules, 393
 Jupyter cell, 393, 401
 one-dimensional operation
 dilation, 262
 input vector, 259, 260
 output vector, 260, 261
 standard values, 394, 395
 TCN class, 399, 400, 406
 testing function, 404, 405, 407, 408
 training function, 402
 training/testing sets, 396–398
 TensorBoard
 command prompt, 354
 graph, 357, 358
 MNIST data set, 353
 parameters, 352, 353
 val_acc/val_loss, 356
 TensorFlow, 84, 113, 121, 320
 train_test_split function, 46, 274
 Transfer learning, 259
 Transportation sector, 306, 307

U

Unit length scaling, 91
 Unsupervised anomaly
 detection, 19, 34
 Upsampling, 285, 337

INDEX

V, W, X, Y, Z

Variational autoencoder

- anomalies, [175](#)
- confusion matrix, [173](#), [174](#)
- definition, [163](#)
- distribution code, [169](#)

import packages, [165](#), [166](#)

neural network, [164](#), [170](#), [171](#)

Pandas dataframe, [168](#)

results via confusion matrix, [167](#)

training process, [172](#), [173](#), [176](#), [177](#)

Video surveillance, [312](#), [313](#)